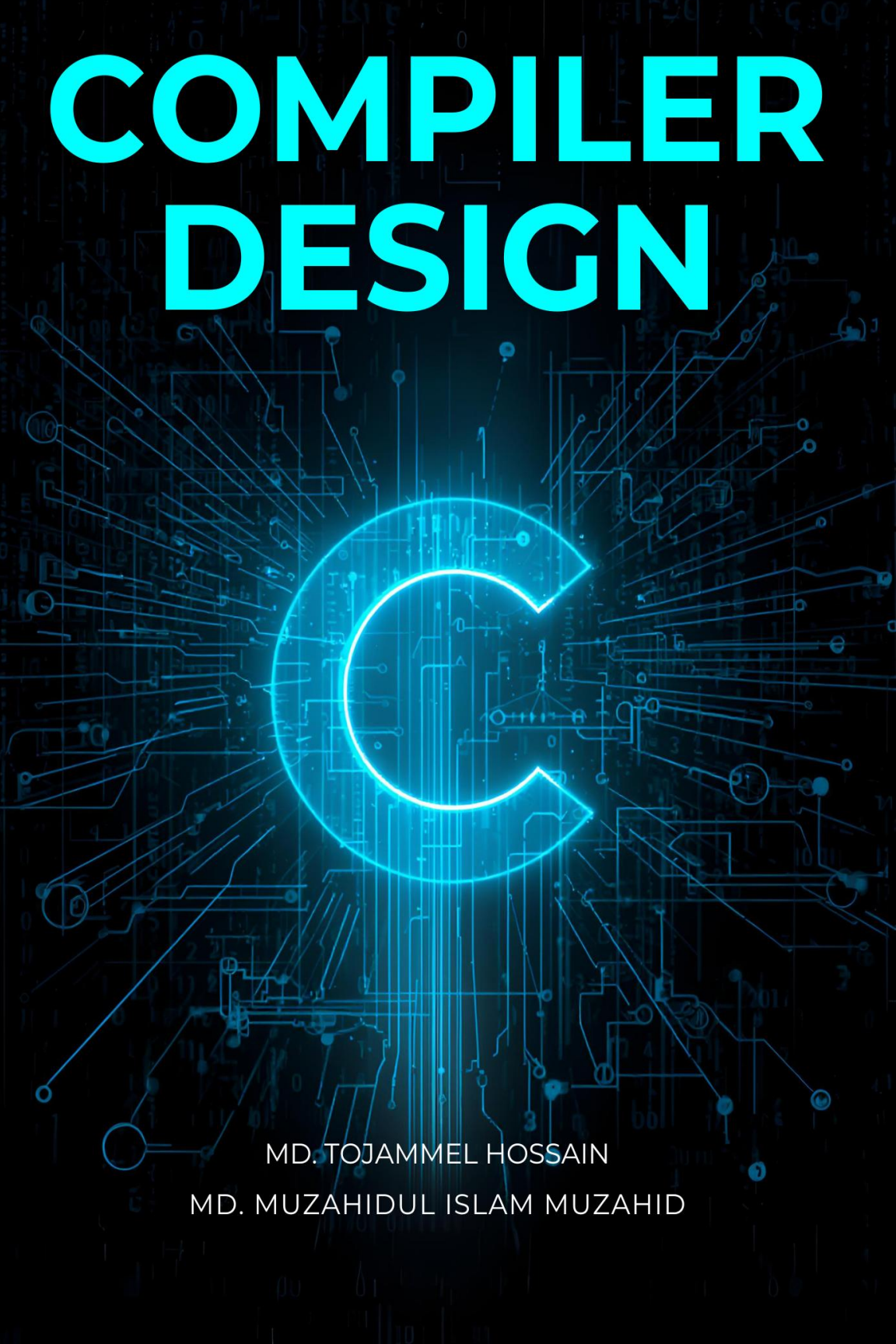
COMPILER DESIGN



MD. TOJAMMEL HOSSAIN

MD. MUZAHIDUL ISLAM MUZAHID

Compiler Design

A Practical Approach to Compiler Construction

By
Md. Tojammel Hossain
Md. Muzahidul Islam Muzahid

Edited By
Mushfiqur Rahman
Assistant Professor
Department of CSE
Daffodil International University

First Edition 2026
Bangladesh

Compiler Design

Copyright © 2026 by

MD. Tojammel Hossain
MD. Muzahidul Islam Muzahid

All rights reserved.

This work is prepared for academic and personal use.
No part of this document may be reproduced or distributed
without proper permission of the authors.

First Edition: 2026
Printed in Bangladesh

PREFACE

Writing this book on Compiler Design has been a journey driven by both curiosity and necessity. During my academic studies, I realized that many existing resources either focus too heavily on theory or are difficult for beginners to understand. This inspired me to present compiler concepts in a clear, structured, and practical manner, enabling students to grasp even complex topics with confidence.

The content of this book is designed to guide readers through the complete process of compiler construction, starting from fundamental concepts such as lexical and syntax analysis, and gradually progressing to advanced topics like code optimization and code generation. Along with theoretical explanations, the book includes diagrams, step-by-step examples, and implementations in C, helping readers connect concepts with real-world applications.

This book is primarily intended for undergraduate students of Computer Science and Engineering, especially those studying compiler design for the first time. It can also serve as a useful resource for anyone interested in understanding how programming languages are translated and executed within a computer system.

I would like to express my sincere gratitude to my respected teacher, Mushfiqur Rahman, Assistant Professor, Department of CSE, Daffodil International University, for his invaluable guidance, continuous support, and inspiration. I am also thankful to my peers and well-wishers for their encouragement throughout this journey.

TABLE OF CONTENTS

PREFACE 4
CHAPTER 1 2
INTRODUCTION TO COMPILERS 2
1.1 What is a Compiler 2
1.2 Why Compilers are Important 3
1.3 Compiler vs Interpreter 4
1.4 Language Processing System 5
1.5 Phases of a Compiler 6
1.6 Overview of Compilation Process 8
1.7 Errors in Compilation 8
Exercise 10
CHAPTER 2 12
COMPILER STRUCTURE & TOOLS 12
2.1 Structure of a Compiler 12
2.2 Frontend and Backend 13
2.3 Symbol Table 15
2.4 Error Handling Mechanisms 16
2.5 Compiler Construction Tools 17
2.6 Overview of LEX 18
Exercise 20

CHAPTER 3 23
LEXICAL ANALYSIS 23
3.1 Introduction to Lexical Analysis 23
3.2 Tokens, Lexemes and Patterns 23
3.3 Role of Lexical Analyzer 25
3.4 Input Buffering 27
3.5 Lexical Errors 29
3.6 Finite Automata Basics 30
3.7 Implementation of Lexical Analyzer in C 32
3.8 Mini Project: Scanner Design 34
Exercise 36
CHAPTER 4 38
REGULAR EXPRESSIONS & FINITE AUTOMATA 38
4.1 Regular Expressions 38
4.2 Finite Automata 39
4.3 NFA and DFA 40
4.4 Conversion: RE → NFA → DFA 42
4.5 DFA Design for Tokens 43
Exercise 46
CHAPTER 5 48
SYNTAX ANALYSIS BASICS 48
5.1 Introduction to Parsing 48
5.2 Context-Free Grammar (CFG) 49
5.3 Parse Trees 50
5.4 Ambiguous Grammar 51

Exercise 53
CHAPTER 6 54
TOP-DOWN PARSING 54
6.1 Introduction to Top-Down Parsing 54
6.2 Recursive Descent Parsing 55
6.3 Predictive Parsing 57
6.4 FIRST and FOLLOW 58
6.5 LL(1) Parsing Table 59
Exercise 61
CHAPTER 7 62
BOTTOM-UP PARSING 62
7.1 Introduction to Bottom-Up Parsing 62
7.2 Shift-Reduce Parsing 64
7.3 Handle Concept 65
7.4 LR Parsing Overview 67
7.5 SLR, LALR, CLR Parsers 68
Exercise 70
CHAPTER 8 72
SEMANTIC ANALYSIS 72
8.1 Introduction to Semantic Analysis 72
8.2 Type Checking 73
8.3 Symbol Table Usage 74
8.4 Scope Management 75
8.5 Type Conversion 76
8.6 Semantic Errors 77

8.7 Syntax Directed Translation 78
Exercise 80
CHAPTER 9 82
INTERMEDIATE CODE 82
9.1 Introduction to Intermediate Representation 82
9.2 Syntax Trees 83
9.3 Directed Acyclic Graph (DAG) 84
9.4 Three Address Code (TAC) 85
9.5 Quadruples, Triples, Indirect Triples 86
Exercise 88
CHAPTER 10 90
CODE OPTIMIZATION 90
10.1 Introduction to Optimization 90
10.2 Goals of Optimization 91
10.3 Types of Optimizations 92
10.4 Basic Blocks 93
10.5 Optimization Techniques 94
10.5.1 Constant Folding 94
10.5.2 Dead Code Elimination 95
10.5.3 Common Subexpression Elimination 96
10.5.4 Copy Propagation 96
10.5.5 Loop Optimization 97
Exercise 99
CHAPTER 11 101
CODE GENERATION 101

11.1 Introduction to Code Generation 101
11.2 Role of Code Generator 102
11.3 Input and Output of Code Generator 103
11.4 Register Allocation 104
11.5 Instruction Selection 105
11.6 Code Generation Examples 106
Exercise 108
CHAPTER 12 110
MINI COMPILER DESIGN 110
12.1 Introduction to Mini Compiler 110
12.2 Compiler Architecture 111
12.3 Step-by-Step Compilation Process 113
12.4 Grammar Design 114
12.5 Example Walkthrough 114
Exercise 117
CHAPTER 13 118
LEX and YACC 118
13.1 Introduction to LEX 118
13.2 Structure of LEX Program 118
13.3 LEX Examples 119
13.4 Introduction to YACC 121
13.5 Structure of YACC Program 122
13.6 YACC Examples 123
13.7 LEX & YACC Integration 124
13.8 FLEX and BISON 125

Exercise 127
CHAPTER 14.....128
BUILDING A FULL COMPILER 128
14.1 Lexical Analyzer Implementation 128
14.2 Parser Implementation.....128
14.3 Integrating Lexical + Parser 129
14.4 Handling Multiple Statements.....130
14.5 Compiler Limitations and Improvements.....131
Exercise 134
References 135

Part I │ Foundations of Compiler Design

CHAPTER 1

INTRODUCTION TO COMPILERS

1.1 What is a Compiler

A compiler is a system program that translates a high-level programming language into a lower-level language, usually machine code or assembly language, so that it can be executed by a computer.

High-level languages such as C, C++, and Java are designed to be human-readable, but computers cannot understand them directly. A compiler acts as a bridge between the human-readable code and the machine-executable instructions.

For example, consider the statement:

int sum = a + b;

A compiler converts this high-level instruction into a sequence of machine-level instructions that the processor can execute.

The basic concept of a compiler is illustrated in **Figure 1.1**, where the source program is transformed into target code.

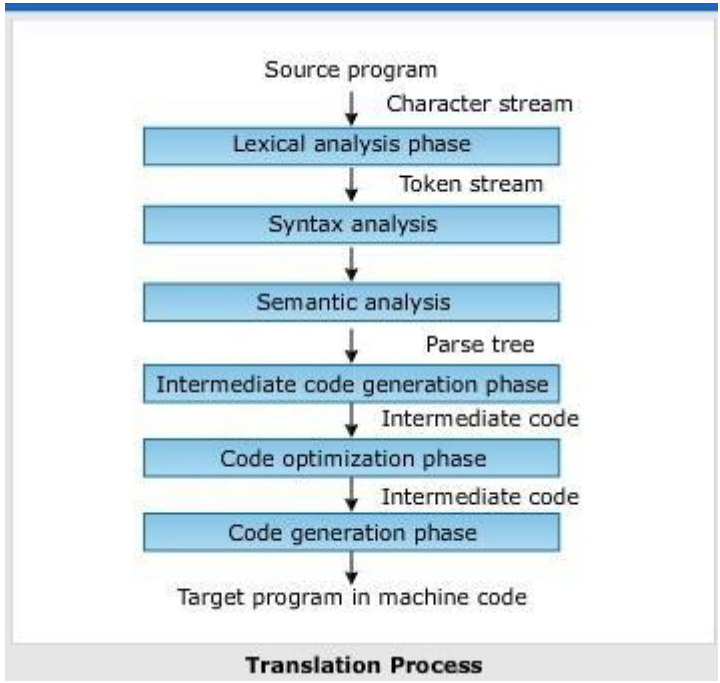


Figure 1.1: Basic Function of a Compiler

As shown in Figure 1.1, the compiler reads the source program and produces equivalent target code.

1.2 Why Compilers are Important

Compilers play a crucial role in software development because they enable programs written in high-level languages to run on computers.

Without compilers, programmers would have to write programs directly in machine language, which is complex and error-prone. Compilers make programming easier by allowing developers to focus on logic rather than hardware details.

For example, a programmer can write a simple program:

```
printf("Hello World");
```

The compiler converts this into multiple machine instructions that handle memory, input/output, and execution.

Compilers also help in detecting errors, improving program efficiency, and optimizing code. Therefore, they are essential tools in modern computing systems.

1.3 Compiler vs Interpreter

A compiler and an interpreter are both used to execute programs written in high-level languages, but they work differently.

A compiler translates the entire program at once and produces machine code before execution begins. Once compiled, the program can be executed multiple times without recompilation.

An interpreter, on the other hand, translates and executes the program line by line. It does not produce a separate machine code file.

Consider the following example:

```
x = 5 + 3;
```

In a compiled language, this statement is translated into machine code before execution. In an interpreted language, the statement is translated and executed immediately.

The difference between these approaches is illustrated in **Figure 1.2**, where compilation and interpretation processes are compared.

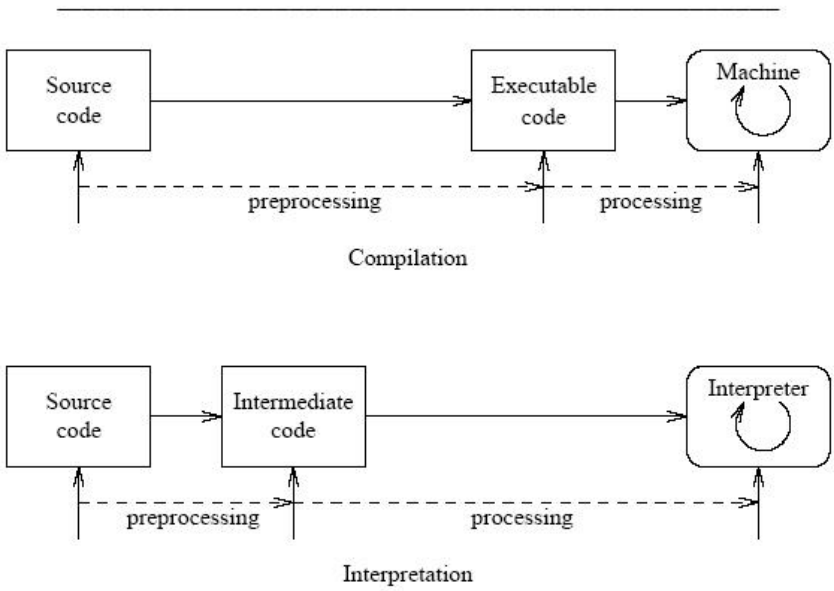


Figure 1.2: Compiler vs Interpreter

As shown in Figure 1.2, compilers generate target code in advance, while interpreters execute instructions directly.

1.4 Language Processing System

A language processing system is a collection of tools used to convert a source program into executable form. It includes preprocessors, compilers, assemblers, and linkers.

The preprocessing stage handles tasks such as macro expansion and file inclusion. The compiler then translates the preprocessed code into assembly or intermediate code. The assembler converts assembly code into machine code, and the linker combines multiple object files into a final executable program.

The complete language processing system is illustrated in **Figure 1.3**, where each stage contributes to producing the final executable file.

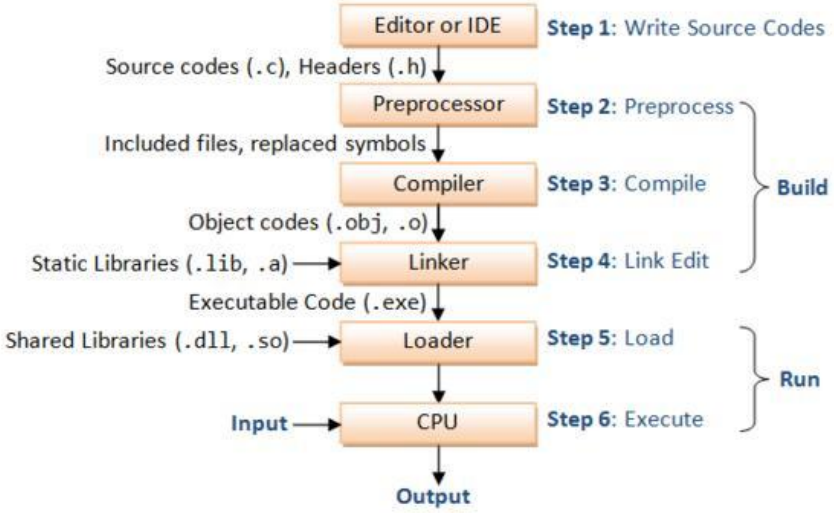


Figure 1.3: Language Processing System

As shown in Figure 1.3, multiple tools work together to convert a program into executable form.

1.5 Phases of a Compiler

A compiler works through several phases, each responsible for a specific task in the translation process.

The process begins with lexical analysis, where the input is divided into tokens. This is followed by syntax analysis, where the structure of the tokens is checked. Semantic analysis then verifies the meaning of the program.

After that, intermediate code generation produces a machine-independent representation. Code optimization improves efficiency, and finally, code generation produces machine instructions.

These phases are illustrated in **Figure 1.4**, where the compiler is shown as a sequence of stages.

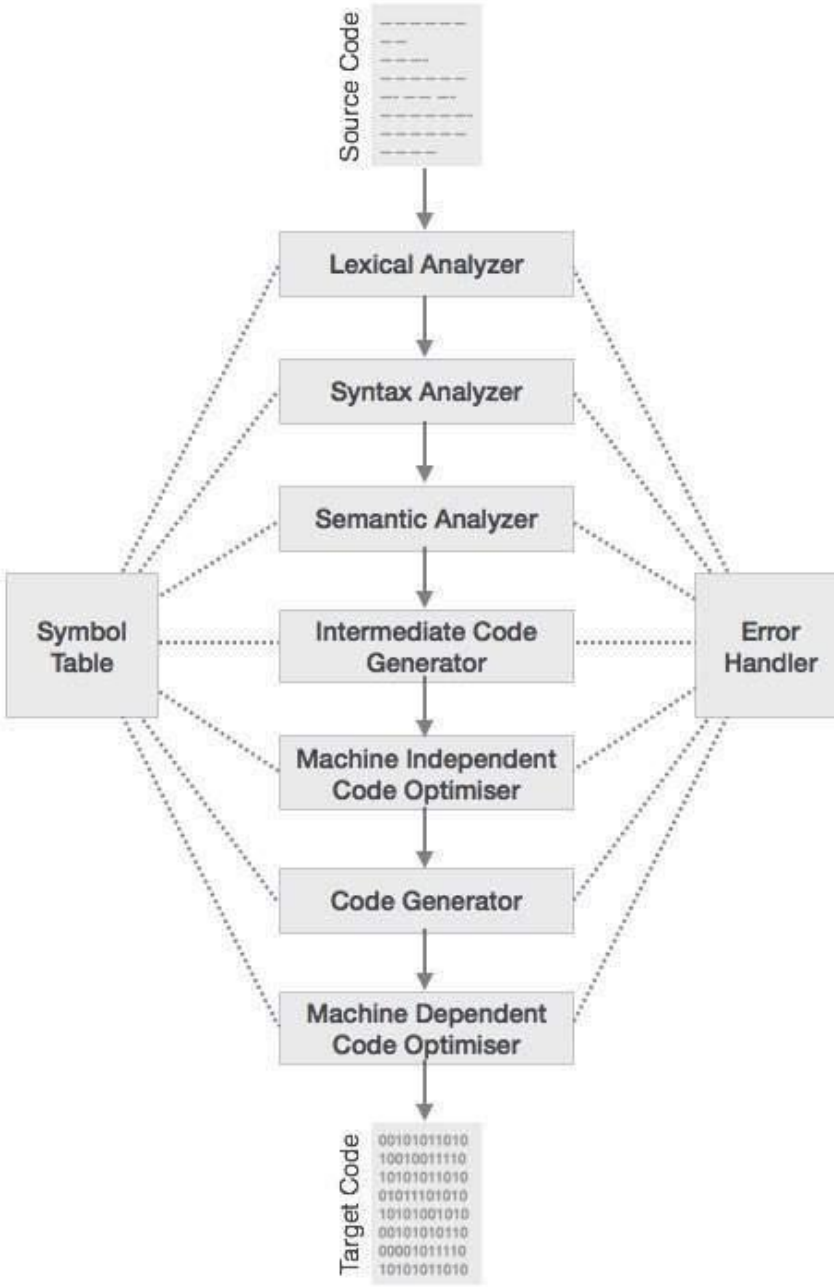


Figure 1.4: Phases of a Compiler

As shown in Figure 1.4, each phase transforms the program into a more refined form.

1.6 Overview of Compilation Process

The compilation process is a step-by-step transformation of a source program into executable code. Each phase takes input from the previous phase and produces output for the next phase.

Consider the input:

```
result = x + y * 2;
```

During lexical analysis, this statement is divided into tokens. Syntax analysis checks whether the expression is valid. Semantic analysis verifies that variables are properly defined and used.

Intermediate code generation converts the expression into a simpler form:

```
t1 = y * 2

t2 = x + t1

result = t2
```

Finally, code generation converts this into machine-level instructions.

1.7 Errors in Compilation

Errors in a program can occur at different stages of compilation. These errors are classified into lexical, syntax, and semantic errors.

A lexical error occurs when invalid characters or patterns are used. For example:

```
int $x;
```

Here, the symbol $ is not valid in identifiers.

A syntax error occurs when the structure of the program is incorrect:

```
int x = ;
```

This statement is incomplete and violates grammar rules.

A semantic error occurs when the program is logically incorrect:

```
int x;

x = "hello";
```

In this case, a string value is assigned to an integer variable.

These types of errors are illustrated in **Figure 1.5**, where different error categories are shown.

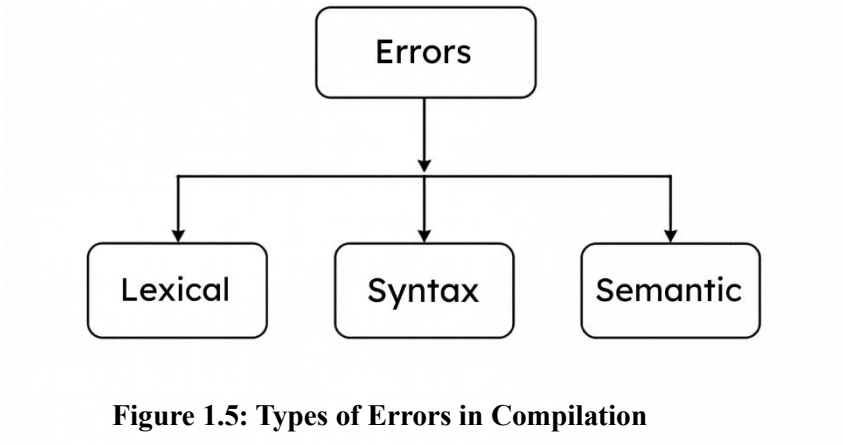


Figure 1.5: Types of Errors in Compilation

As shown in Figure 1.5, each phase of the compiler is responsible for detecting specific types of errors.

EXERCISE

1. Consider the following high-level statement:

```
int result = a + b;
```

Explain how a compiler translates this statement into machine code. Clearly describe each step involved in the translation process.

2. A programming language can be executed using either a compiler or an interpreter.
   Given a program that calculates factorial, explain how it would be executed differently using a compiler and an interpreter. Also discuss advantages of each approach.

3. The following diagram represents a language processing system:

```
Source Program → Preprocessor → Compiler → Assembler
→ Linker → Executable
```

Explain each component in the diagram and describe its role in generating the final executable program.

4. Consider the compilation phases shown below:

```
Lexical → Syntax → Semantic → Intermediate →
Optimization → Code Generation
```

Explain how each phase processes the input using the expression:

```
x = y + 5 * z;
```

5. A programmer writes the following code:

```
int                                                    x;
x = "hello";
```

Identify the type of error and explain at which compilation phase it will be detected and why.

6. Given the following incorrect code:

```
int a = ;
```

Explain how a compiler detects this error. Which phase identifies it and what type of error is it?

7. Explain the role of lexical analysis using the following input:

```
sum = value1 + 20;
```

Show the tokens generated and explain the transformation process.

8. A compiler must handle multiple errors in a program. Explain how error handling is performed during compilation with suitable examples.

9. Explain the difference between source code and target code using a real example. Also describe how intermediate code helps in this transformation.

10. Consider a simple program with multiple statements. Explain how the compiler processes the entire program step by step.

CHAPTER 2

COMPILER STRUCTURE & TOOLS

2.1 Structure of a Compiler

A compiler is not a single program but a combination of several components that work together to translate a source program into target code. These components are organized in a structured way so that each part performs a specific task.

The compiler structure can be broadly divided into phases such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. Each phase processes the output of the previous phase and produces input for the next.

For example, consider the statement:

```
x = a + b * 2;
```

The lexical analyzer converts this into tokens, the syntax analyzer checks the structure, and the later stages generate machine instructions.

The overall structure of a compiler is illustrated in **Figure 2.1**, where different phases are connected in sequence.

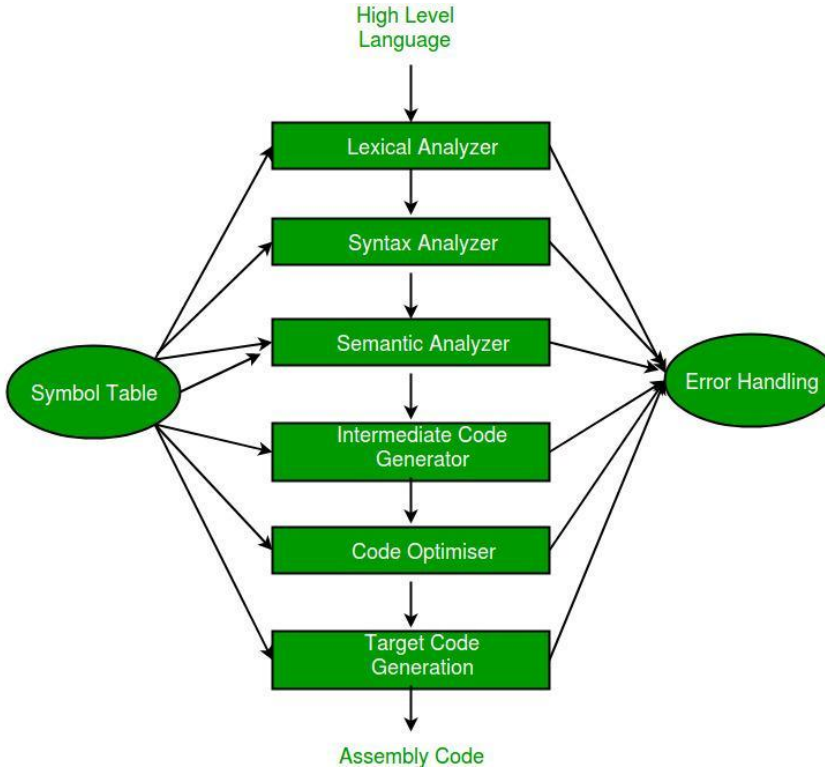


Figure 2.1: Structure of a Compiler

As shown in Figure 2.1, the compiler is organized as a pipeline of phases that gradually transform the program.

2.2 Frontend and Backend

The compiler can be divided into two major parts: the frontend and the backend. These two parts are responsible for different stages of compilation.

The frontend deals with the analysis of the source program. It includes lexical analysis, syntax analysis, and semantic analysis. The main goal of the frontend is to understand the program and check whether it is correct.

The backend is responsible for generating target code. It includes intermediate code generation, optimization, and final code generation. The backend focuses on producing efficient machine code.

For example, in the statement:

```
result = p + q;
```

the frontend ensures that the expression is valid, while the backend converts it into machine instructions such as:

```
MOV R1, p

ADD R1, q

MOV result, R1
```

The relationship between frontend and backend is illustrated in **Figure 2.2**, where the compiler is divided into two major sections.

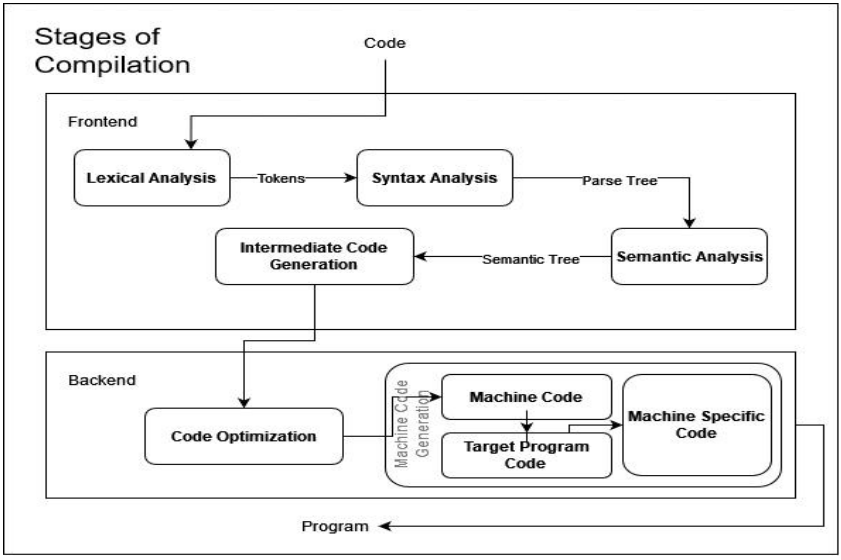


Figure 2.2: Frontend and Backend of a Compiler

As shown in Figure 2.2, the frontend analyzes the program, while the backend generates optimized machine code.

2.3 Symbol Table

The symbol table is a data structure used by the compiler to store information about identifiers such as variables, functions, and constants.

Each entry in the symbol table contains details such as the name of the identifier, its type, scope, and memory location. This information is used throughout different phases of the compiler.

For example, consider the code:

```
int a;

float b;
```

The symbol table will store entries for a and b along with their types.

During semantic analysis, if the program uses an undeclared variable, the compiler checks the symbol table and reports an error.

The working of the symbol table is illustrated in **Figure 2.3**, where identifiers are stored and accessed during compilation.

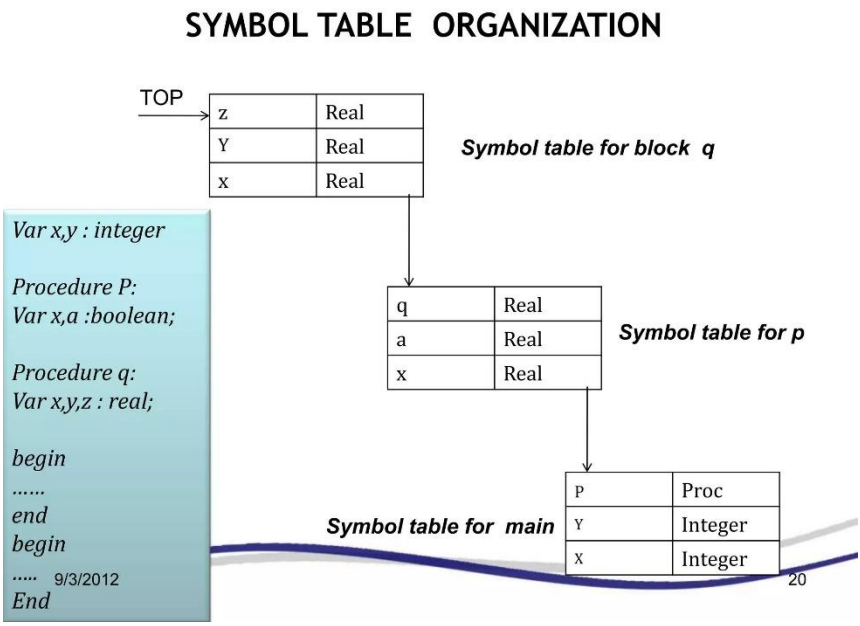


Figure 2.3: Symbol Table in Compiler

As shown in Figure 2.3, the symbol table acts as a central storage for all identifiers used in the program.

2.4 Error Handling Mechanisms

Error handling is an important part of compiler design. The compiler must detect errors and provide meaningful messages to the programmer.

Errors can occur in different phases. Lexical errors occur when invalid characters are used, syntax errors occur when grammar rules are violated, and semantic errors occur when the program is logically incorrect.

Consider the following examples:

```
int x = ;
```

This statement contains a syntax error because the expression is incomplete.

```
int y;

y = "text";
```

This statement contains a semantic error because a string is assigned to an integer.

A good compiler not only detects errors but also tries to recover from them and continue compilation. This allows the programmer to see multiple errors in one run.

2.5 Compiler Construction Tools

Compiler construction tools are software tools that help in building different parts of a compiler automatically. These tools reduce development time and improve accuracy.

Some tools are used for lexical analysis, while others are used for syntax analysis or code generation.

For example, instead of writing a lexical analyzer manually, a tool like LEX can be used to generate it automatically from regular expressions.

Similarly, parser generators can be used to create syntax analyzers from grammar rules.

The role of compiler construction tools is illustrated in **Figure 2.4**, where different tools are used for different phases.

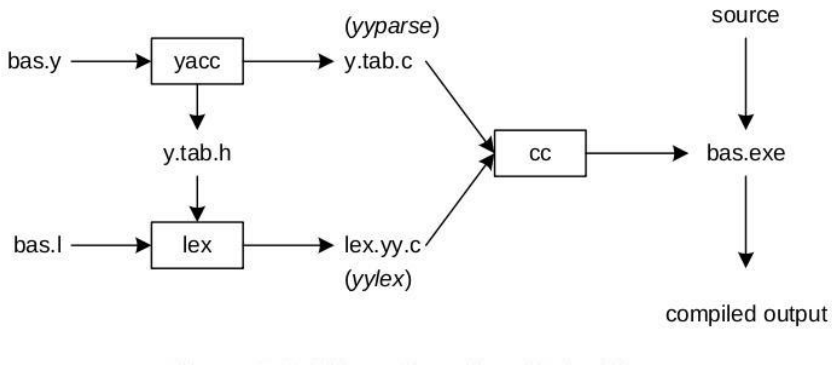


Figure 2.4: Compiler Construction Tools Lex/YACC

As shown in Figure 2.4, tools simplify the process of building compilers by automating various tasks.

2.6 Overview of LEX

LEX is a widely used tool for generating lexical analyzers. It allows the programmer to define token patterns using regular expressions, and it generates a scanner in C language.

Instead of manually writing code to identify tokens, the programmer specifies patterns, and LEX automatically creates the corresponding scanning program.

For example, a pattern like:

```
[a-zA-Z]+
```

can be used to identify identifiers. Similarly:

```
[0-9]+
```

can be used to recognize numbers.

When LEX processes input such as:

```
x + 10
```

it identifies x as an identifier, + as an operator, and 10 as a number.

The working of LEX is illustrated in **Figure 2.5**, where patterns are matched against input to produce tokens.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Figure 2.5: Overview of LEX Tool

As shown in Figure 2.5, LEX simplifies lexical analysis by automatically generating a scanner based on pattern definitions.

EXERCISE

1. Consider the compiler structure shown below:

```
Source → Lexical → Syntax → Semantic → Code Generation
```

Explain how data flows through this structure using the statement:

```
a = b + c * d;
```

2. A compiler is divided into frontend and backend. Explain the role of each part using an example and describe how they interact with each other.

3. Consider the following program:

```
int                                                    a;
float                                                  b;
a = b;
```

Explain how the symbol table is created and updated during compilation.

4. A compiler detects errors at different stages. Given the following statements, identify the type of error and phase:

```
int              x              =              ;
y = 5;
```

Explain how error handling mechanisms work.

5. Explain the importance of compiler construction tools using real examples such as LEX and YACC.

6. Consider a lexical analyzer generated using LEX. Explain how patterns are defined and how tokens are generated.

7. Explain how different phases of a compiler communicate with each other using an example.
8. Describe how symbol table entries are created, updated, and accessed during compilation.
9. Consider a simple arithmetic expression.
   Explain how the compiler architecture processes it step by step.
10. Explain how automation tools reduce the complexity of compiler construction with examples.

Part II | Lexical Analysis

CHAPTER 3

LEXICAL ANALYSIS

3.1 Introduction to Lexical Analysis

Lexical Analysis is the first phase of a compiler. In this phase, the source code is read character by character and converted into meaningful units called tokens.

The component that performs this task is called the **lexical analyzer** or **scanner**.

The main goal of lexical analysis is to simplify the source program so that the next phases of the compiler can process it easily.

For example, consider the following statement:

```
total = x + 25;
```

The lexical analyzer breaks this into tokens such as identifier (total), assignment operator (=), identifier (x), operator (+), and number (25). Each of these is treated as a separate unit.

Thus, lexical analysis acts as a bridge between the source code and the syntax analysis phase.

3.2 Tokens, Lexemes and Patterns

In lexical analysis, three important terms are used: **tokens**, **lexemes**, and **patterns**. In **Table 3.1**, there are specification for Lexical Analysis.

Token:

A token is a category or type of a string in the source code. It represents a group of similar lexemes.

Examples of tokens include:

- Keyword
- Identifier
- Operator
- Constant
- Separator

Lexeme

A lexeme is the actual sequence of characters in the source code that matches a token.

Example:

int count = 5;

Here:

- int is a lexeme (token: keyword)
- count is a lexeme (token: identifier)
- 5 is a lexeme (token: constant)

Pattern

A pattern is a rule that describes the form of a lexeme.

Examples:

- Identifier → letter followed by letters or digits
- Number → sequence of digits

Table 3.1: Token Specification for Lexical Analysis

Token Type	Lexeme Example	Pattern
Keyword	int	predefined word
Identifier	count	letter (letter
Constant	100	digit+
Operator	+	arithmetic symbol

3.3 Role of Lexical Analyzer

The lexical analyzer plays a crucial role in the compilation process. It is the **first phase of the compiler**, responsible for scanning the source program and converting it into a sequence of meaningful tokens that can be used by the syntax analyzer.

Main Responsibilities

The lexical analyzer performs the following functions:

- **Reading the source code:**
  It scans the input program character by character.
- **Removing white spaces and comments:**
  Unnecessary elements such as spaces, tabs, and comments are ignored since they do not affect the program logic.
- **Identifying tokens:**
  It groups characters into valid tokens such as identifiers, keywords, operators, and numbers.
- **Passing tokens to the parser:**
  The generated tokens are sent to the syntax analyzer for further processing.

- **Maintaining symbol table entries:**
  It stores information about identifiers (such as variable names) in the symbol table.
- **Detecting lexical errors:**
  It identifies invalid patterns such as illegal symbols or malformed identifiers.

Working Flow

The operation of the lexical analyzer can be represented as:

```
Source Code → Lexical Analyzer → Tokens → Syntax Analyzer
```

The lexical analyzer acts as a bridge between the source program and the parser**.**

Example

Consider the following source code:

```
a = b + 5;
```

The lexical analyzer produces the following tokens:

```
IDENTIFIER    (a)
ASSIGN        (=)
IDENTIFIER    (b)
PLUS          (+)
NUMBER        (5)
SEMICOLON     (;)
```

Each token represents a meaningful unit of the program.

Important Note:

The lexical analyzer does not check the grammar or structure of the program. Its primary task is only to break the input into tokens.

The responsibility of checking whether the tokens form a valid structure lies with the **syntax analyzer (parser)**.

3.4 Input Buffering

Reading characters one by one directly from the source file is inefficient because each read operation requires system-level input/output (I/O). This can significantly slow down the compilation process, especially for large programs.

To overcome this problem, compilers use a technique called **input buffering**, which improves the speed of lexical analysis.

Why Input Buffering is Required

Input buffering is used to:

- Reduce the number of I/O operations.
- Increase processing speed.
- Allow the compiler to look ahead in the input.

Instead of reading one character at a time, the compiler reads a block of characters into memory and processes them from there.

Two-Buffer Scheme

The most commonly used buffering technique is the **two-buffer scheme**.

In this approach:

- Two memory buffers are used
- Each buffer stores a block of input

- While one buffer is being processed, the other is refilled

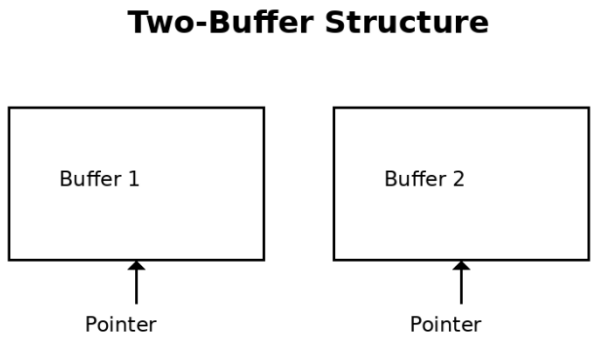


Figure 3.1: Two-Buffer Structure

In **Figure 3.1**, where each buffer has a pointer that moves forward as characters are read. When one buffer is exhausted, the compiler switches to the other buffer.

Sentinel Technique

A special character called a **sentinel** is placed at the end of each buffer.

- It marks the end of the buffer
- It eliminates the need for repeated boundary checking

This technique improves efficiency because the compiler does not need to check whether it has reached the end of the buffer after every character.

Advantages of Input Buffering

- Faster lexical analysis
- Efficient use of memory
- Supports lookahead
- Reduces overhead

3.5 Lexical Errors

Lexical errors occur when the lexical analyzer encounters a sequence of characters that does not match any valid token pattern.

These errors are detected during the first phase of compilation.

Common Types of Lexical Errors

Lexical errors can occur due to:

- Invalid characters
- Incorrect identifier format
- Unterminated strings
- Invalid numeric constants

Examples of Lexical Errors

```
//C


int $num = 10;
```

Error: $ is not a valid character in identifiers

```
//C


int 9value;
```

Error: Identifiers cannot start with a digit

```
//C


"Hello
```

Error: Missing closing quotation mark

Error Detection and Handling

The lexical analyzer should:

1. Detect the error
2. Report a meaningful message
3. Continue scanning the rest of the input

Example Error Message

```
Lexical Error: Invalid symbol '$' at line 1
```

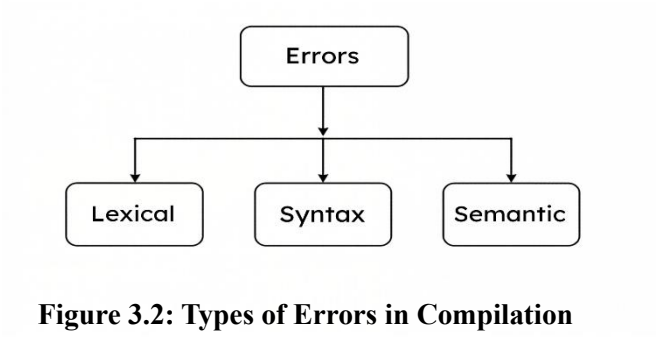


Figure 3.2: Types of Errors in Compilation

3.6 Finite Automata Basics

Finite Automata are mathematical models used to recognize patterns in input strings. They are widely used in lexical analysis to identify tokens based on patterns.

Types of Finite Automata

There are two main types:

1. Deterministic Finite Automaton (DFA)
2. Non-deterministic Finite Automaton (NFA)

Deterministic Finite Automaton (DFA)

In DFA:

- Each state has exactly one transition for each input symbol
- There is no ambiguity
- It is easier to implement

Non-deterministic Finite Automaton (NFA)

In NFA:

- A state may have multiple transitions
- It may include epsilon (ε) transitions
- It is more flexible but harder to implement directly

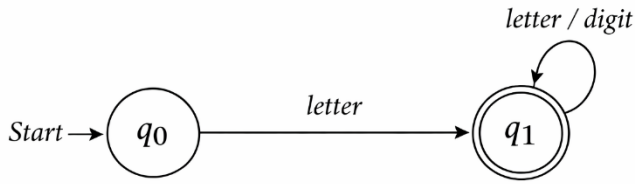
Components of Finite Automata

A finite automaton consists of:

- A set of states
- Input symbols
- Transition function
- Start state
- Final (accepting) states

```
Example: Identifier Recognition
```

An identifier must start with a letter and can be followed by letters or digits.



Where:

- q0 = start state
- q1 = accepting state / final state (double circle)

Figure 3.3: DFA for Identifier

Role in Lexical Analysis

Finite automata help in:

- Recognizing tokens
- Matching patterns
- Building lexical analyzers

3.7 Implementation of Lexical Analyzer in C

A lexical analyzer can be implemented using a programming language such as C. The following program demonstrates a simple scanner that classifies characters.

Program 3.1: Simple Lexical Analyzer

main.c

```c
#include <stdio.h>
#include <ctype.h>

int main() {
    char input[100];
    int i = 0;

    printf("Enter a string: ");
    scanf("%s", input);

    while (input[i] != '\0') {
        if (isalpha(input[i])) {
            printf("LETTER: %c\n", input[i]);
        }
        else if (isdigit(input[i])) {
            printf("DIGIT: %c\n", input[i]);
        }
        else {
            printf("SPECIAL SYMBOL: %c\n", input[i]);
        }
        i++;
    }

    return 0;
}
```

If we input: AB5+

Output

```
Enter a string: AB5+
LETTER: A
LETTER: B
DIGIT: 5
SPECIAL SYMBOL: +


=== Code Execution Successful ===
```

Explanation of the Program

- isalpha() checks whether the character is a letter
- isdigit() checks whether the character is a digit

- All other characters are treated as special symbols

3.8 Mini Project: Scanner Design

In this section, we design a simple lexical analyzer (scanner) that can identify different types of tokens.

Objective

The scanner should identify:

- Keywords
- Identifiers
- Numbers
- Unknown tokens

Program 3.2: Pseudocode of a Simple Scanner Design

```
FUNCTION isKeyword(str)
    keywords = ["int", "float", "if", "else", "while",
"return"]
    FOR i = 0 to length(keywords) - 1
        IF str == keywords[i] THEN
            RETURN 1   // it is a keyword
        END IF
    END FO
    RETURN 0   // not a keyword
END FUNCTION
```

Sample Input and Output

Input:

```
int
```

Output:

```
Keyword
```

Input:

```
value
```

Output:

```
Identifier
```

Explanation

- The function isKeyword() checks if the input matches predefined keywords
- If not, the program checks whether it is an identifier or a number
- Otherwise, it classifies it as an unknown token

This mini project demonstrates how a lexical analyzer identifies tokens and categorizes them. It provides a basic understanding of how scanners work in real compilers.

EXERCISE

1. Consider the input:

```
x = y + 25;
```

Explain how lexical analysis converts this into tokens. Show each step of token generation.

2. Given the statement:

```
int count = 100;
```

Identify tokens, lexemes, and patterns, and explain their relationship.

3. The following flow represents lexical analysis:

```
Source Code → Lexical Analyzer → Tokens → Parser
```

Explain the role of lexical analyzer in this process with example.

4. Consider a two-buffer input system. Explain how input buffering improves efficiency with the help of a diagram.

5. Identify lexical errors in the following code and explain:

```
int $var;
num = 12a;
```

6. Design a DFA for recognizing identifiers and explain each state transition.

7. Explain the difference between DFA and NFA using an example from lexical analysis.

8. Consider the following C code:

```
if (isalpha(ch))
    printf("LETTER");
```

Explain how a lexical analyzer uses such logic to identify tokens.

9. Design a simple scanner that identifies identifiers and numbers. Explain how it processes input step by step.

10. Given the input:

```
a1 = b2 + 300;
```

Explain how the scanner recognizes each token and classifies them.

CHAPTER 4

REGULAR EXPRESSIONS & FINITE AUTOMATA

4.1 Regular Expressions

A regular expression (RE) is a formal way of describing patterns in strings. It is widely used in lexical analysis to define tokens such as identifiers, numbers, and keywords.

In Table 4.1 shown basic symbols of regular expressions that provide a concise and powerful method for specifying sets of strings.

Table 4.1: Basic Symbols in Regular Expressions

Symbol	Meaning
a	single character
ab	concatenation
a | b	union (either a or b)
a*	zero or more occurrences
a+	one or more occurrences
( )	grouping

Examples

- a* → "", a, aa, aaa, ...
- a|b → a or b
- ab → sequence a followed by b

Example: Identifier Pattern

```
letter (letter | digit)*
```

This means:

- first character must be a letter
- followed by letters or digits

Example: Number Pattern

```
digit+
```

Application in Compiler

Regular expressions are used to define:

- identifiers
- constants
- operators
- keywords

```
Pattern → Input String → Match / No Match
```

Figure 4.1: Basic working of regular expression matching.

4.2 Finite Automata

Finite Automata (FA) are mathematical models used to recognize patterns defined by regular expressions.

They are widely used in lexical analyzers to recognize tokens.

Components of Finite Automata

A finite automaton consists of:

- A finite set of states
- Input symbols
- Transition function
- Start state
- One or more final states

Working Principle

The automaton reads input symbols one by one and moves between states according to the transition rules.

If it ends in a final state, the string is accepted.

Example

Consider a language that accepts strings ending with ab.

States: q0, q1, q2
Start: q0
Final: q2

4.3 NFA and DFA

Finite automata are of two types:

- **Non-deterministic Finite Automaton (NFA)**
- **Deterministic Finite Automaton (DFA)**

Non-deterministic Finite Automaton (NFA)

In an NFA:

- A state can have multiple transitions for the same input
- It can move without consuming input (ε-transition)

Example NFA

(q0) --a--> (q1)
(q0) --a--> (q2)

Here, from state q0 with input 'a', it can go to multiple states.

Deterministic Finite Automaton (DFA)

In a DFA:

- Each state has exactly one transition for each input
- No ε-transitions are allowed

Example DFA

```
(q0) --a--> (q1)
(q1) --b--> (q2)
```

Figure 4.2: NFA vs DFA

NFA: Multiple paths possible
DFA: Single path for each input

Table 4.2: Comparison of NFA and DFA

Feature	NFA	DFA
Transitions	Multiple allowed	Single
ε-moves	Allowed	Not allowed
Complexity	Easier to design	Easier to implement
Speed	Slower	Faster

4.4 Conversion: RE → NFA → DFA

Regular expressions are first converted into NFA, and then NFA is converted into DFA for implementation.

Step 1: RE → NFA

Each regular expression can be converted into an NFA using standard construction rules.

Example:

RE: a|b

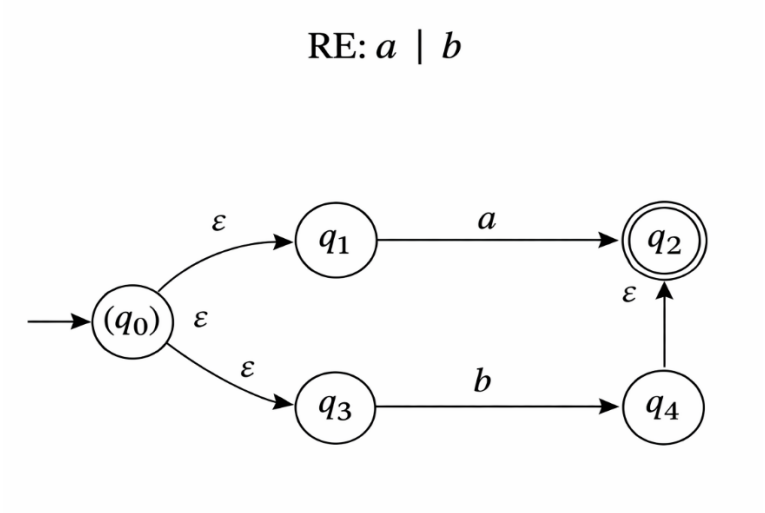


Figure 4.3: NFA representation of RE (a|b).

Step 2: NFA → DFA

NFA is converted into DFA using **subset construction method**.

Steps:

1. Start with ε-closure of initial state
2. Create new DFA states

3. Repeat transitions
4. Mark final states

```
NFA States → DFA State Set
{q0} → A
{q1,q2} → B
```

Figure 4.4: Conversion of NFA states into DFA states.

Importance

- RE is easy to write
- NFA is easy to construct
- DFA is efficient to implement

Thus, conversion ensures both simplicity and efficiency.

4.5 DFA Design for Tokens

In lexical analysis, DFA is used to recognize tokens such as identifiers, numbers, and keywords.

Example: DFA for Identifier

Rules:

- must start with a letter
- followed by letters or digits

```
(q0) --letter--> (q1)
(q1) --letter/digit--> (q1)

Final state: q1
```

Figure 4.5: DFA for recognizing identifiers.

Example: DFA for Integer

```
(q0) --digit--> (q1)
(q1) --digit--> (q1)
```

Combined Token Recognition

A lexical analyzer may use multiple DFAs to recognize:

- identifiers
- numbers
- operators

Program 4.2: Pseudocode for Identifier Recognition using DFA

```
BEGIN
    Read string
    i ← 0
    If string[i] is not a letter then
        Print "Invalid Identifier"
        Exit
    End If
    i ← i + 1
    While string[i] ≠ end of string do
        If string[i] is a letter or digit then
            i ← i + 1
        Else
```

```
            Print "Invalid Identifier"
            Exit
        End If
    End While
    Print "Valid Identifier"
END
```

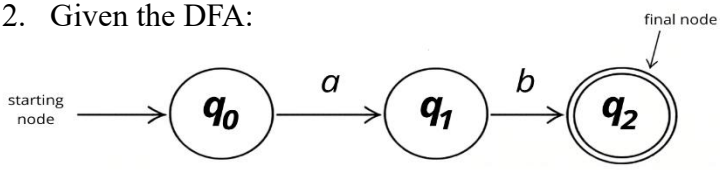
EXERCISE

1. Consider the regular expression:

```
(a|b)*abb
```

Design an NFA for this expression and explain how strings like aabb and babb are accepted.

2. Given the DFA:



Explain whether the string ab and abb are accepted and justify your answer.

3. Convert the following regular expression into NFA and then into DFA:

```
a(b|c)*
```

4. Design a DFA to recognize identifiers consisting of letters followed by digits.
5. Explain the difference between DFA and NFA using a real example from lexical analysis.
6. Show step-by-step conversion of an NFA to DFA.
7. Explain how regular expressions are used to define tokens in lexical analysis.
8. Given the pattern [0-9]+, explain how it is implemented using finite automata.
9. Design a DFA for recognizing strings ending with 01.
10. Explain closure properties of regular expressions with suitable examples.

Part III | Syntax Analysis

CHAPTER 5

SYNTAX ANALYSIS BASICS

5.1 Introduction to Parsing

Syntax analysis, commonly known as parsing, is the second phase of a compiler. After the lexical analyzer converts the source program into a sequence of tokens, the parser examines whether these tokens are arranged according to the grammatical rules of the programming language.

In other words, parsing determines whether the input program has a valid structure. While lexical analysis focuses on identifying individual tokens, syntax analysis focuses on how those tokens are combined to form meaningful statements.

For example, consider the following statement:

```
a = b + 5;
```

The lexical analyzer identifies tokens such as identifiers, operators, and numbers. However, it is the parser that checks whether this sequence of tokens follows the correct syntax of an assignment statement.

The overall working of parsing is illustrated in **Figure 5.1**, where the sequence of tokens generated by the lexical analyzer is passed to the parser, which produces a structured representation such as a parse tree.

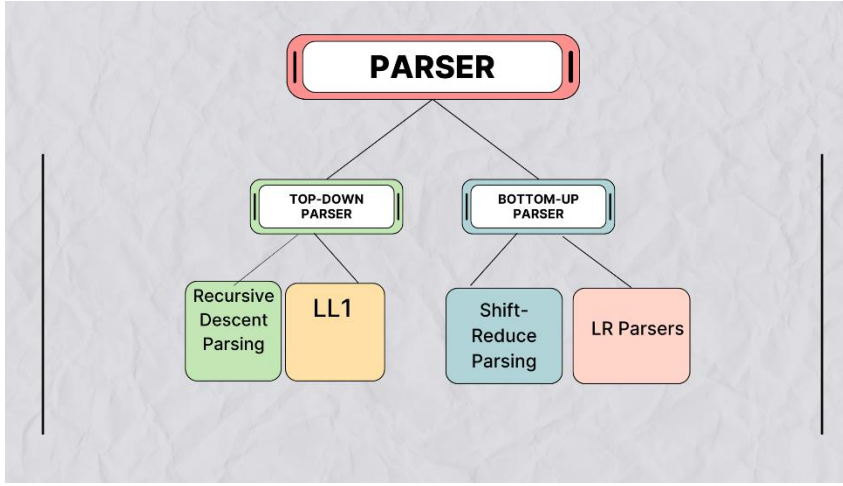


Figure 5.1: Working of Syntax Analysis Phase

As shown in Figure 5.1, the parser acts as a bridge between lexical analysis and semantic analysis. It ensures that the program structure is valid before further processing.

5.2 Context-Free Grammar (CFG)

A context-free grammar (CFG) is used to formally define the syntax of programming languages. It provides a set of rules that describe how valid strings in the language can be formed.

A CFG consists of terminals, non-terminals, production rules, and a start symbol. Terminals represent actual symbols of the language, such as identifiers and operators, while non-terminals represent abstract syntactic categories.

The grammar used for arithmetic expressions is shown in **Table 5.1** below.

Table 5.1: Context-Free Grammar for Arithmetic Expressions

Rule No.	Production Rule
1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → id

In Table 5.1, the symbol E represents an expression, T represents a term, and F represents a factor. The grammar defines how complex expressions can be constructed using simpler components.

Using the rules given in Table 5.1, a valid expression such as:

```
id + id * id
```

can be generated step by step. This process of generating a string from the start symbol is called derivation.

The CFG plays a crucial role in parsing because it provides the foundation on which parsing algorithms are built.

5.3 Parse Trees

A parse tree is a tree representation that shows how a string is derived from the start symbol of a grammar. It visually represents the syntactic structure of the input.

Each internal node of the tree corresponds to a non-terminal, while each leaf node corresponds to a terminal symbol.

Consider the expression:

```
(a + b) * (c - d)
```

Using the grammar defined earlier in Table 5.1, the parse tree for this expression is shown in **Figure 5.2**.

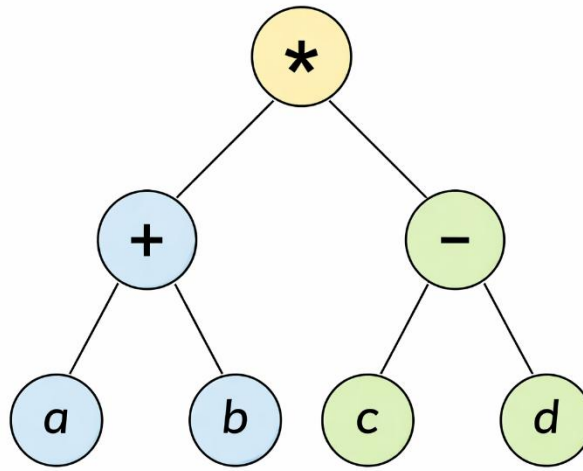


Figure 5.2: Parse Tree for the Expression (a + b) * (c – d)

As illustrated in Figure 5.2, This parse tree shows that * is the main operator at the top, combining two parts: a + b and c − d. The + node connects a and b, and the - node connects c and d. It means first addition and subtraction are done, then their results are multiplied.

The parse tree is important because it clearly shows how the compiler interprets the structure of an expression. It also helps in further stages such as semantic analysis and code generation.

5.4 Ambiguous Grammar

A grammar is said to be ambiguous if a single input string can be derived in more than one way, resulting in multiple parse trees.

This creates confusion because the compiler cannot determine a unique structure for the program.

Consider again the expression:

```
(a + b) * (c – d)
```

If the grammar does not clearly define structure, different interpretations may arise in complex cases, leading to multiple parse trees.

For example, without proper rules, sub-expressions might be grouped differently, causing confusion in evaluation order.

However, in this expression:

- (a + b) and (c − d) are clearly separated using parentheses.
- So, the evaluation order becomes fixed and unambiguous.

This results in a **single parse tree**, where:

- + combines a and b
- − combines c and d
- * combines the results of both

This is desirable in compiler design because it ensures a clear and correct interpretation of the program.

To avoid ambiguity, grammars are carefully designed using **operator precedence**, **associativity**, and **parentheses**, which guarantee a unique parse tree for each valid expression.

Table 5.1 and Figures 5.1 and 5.2 together provide a complete understanding of how a parser works.

EXERCISE

1. Construct a parse tree for the expression:

```
a + b * c * d
```

2. Consider the grammar:

```
E → E + E | id
```

Show that the grammar is ambiguous using an example.

3. Given the expression:

```
x + y * z
```

Show leftmost and rightmost derivations.

4. Explain how syntax analysis detects errors using an incorrect expression.

5. Construct a parse tree for:

```
(a - d) * (c + b)
```

6. Explain the concept of context-free grammar using examples.

7. Differentiate between ambiguous and unambiguous grammar with examples.

8. Explain derivation steps with a suitable grammar.

9. Show how syntax analyzer processes tokens generated by lexical analyzer.

10. Explain the importance of grammar in compiler design.

CHAPTER 6

TOP-DOWN PARSING

6.1 Introduction to Top-Down Parsing

Top-down parsing is a technique used in syntax analysis where the parse tree is constructed starting from the root and proceeding toward the leaves. In this approach, the parser begins with the start symbol of the grammar and tries to derive the input string step by step.

The main idea behind top-down parsing is to predict which production rule should be applied based on the current input symbol. The parser attempts to match the input string with the grammar rules by expanding non-terminals.

For example, consider the expression:

```
id + id * id
```

The parser starts from the start symbol E and tries to derive this string using grammar rules. The step-by-step expansion is guided by the structure of the grammar.

The working of top-down parsing is illustrated in **Figure 6.1**, where the parsing process begins with the start symbol and gradually expands until the input string is matched.

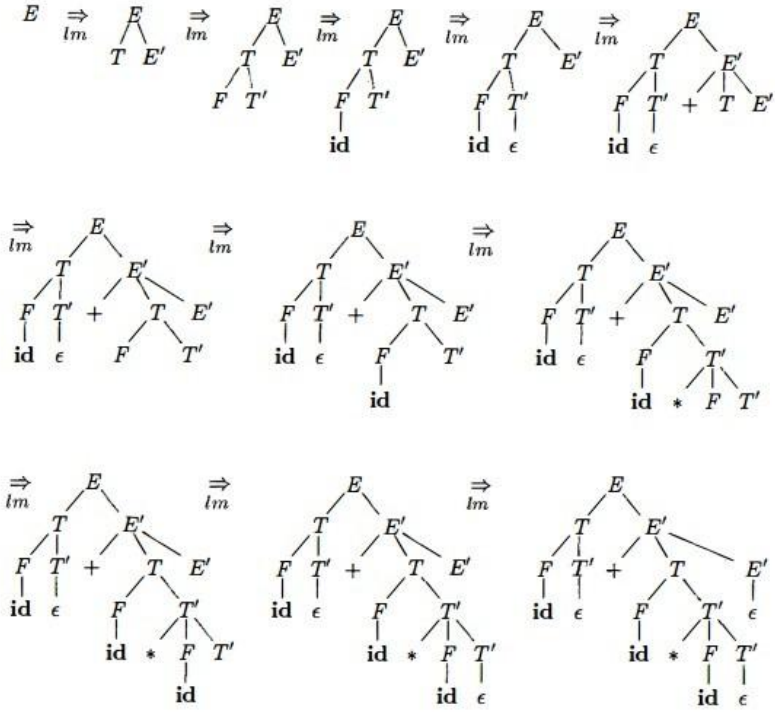


Figure 6.1: Working of Top-Down Parsing

As shown in Figure 6.1, the parser expands non-terminals in a top-down manner until it reaches the terminal symbols that match the input.

6.2 Recursive Descent Parsing

Recursive descent parsing is a top-down parsing technique in which each non-terminal in the grammar is implemented as a function. These functions call each other recursively to process the input string.

This method is simple and easy to implement because it directly follows the grammar rules. However, it may involve backtracking if multiple production rules exist for a non-terminal.

The structure of recursive descent parsing can be understood using the pseudocode shown in **Program 6.1**.

Program 6.1: Pseudocode for Recursive Descent Parsing

```
procedure
E()
    call T()
    while next token is '+'
        match '+'
        call T()


procedure T()
    if next token is 'id'
        match 'id'
    else
        error
```

In Program 6.1, each function corresponds to a non-terminal in the grammar. The parser reads tokens and applies rules accordingly.

Consider the input:

```
id + id
```

The parser calls function E(), which in turn calls T(). After matching the first identifier, it checks for the + operator and processes the next identifier.

The recursive nature of the parsing process is clearly shown in Program 6.1, where functions call each other based on grammar rules.

6.3 Predictive Parsing

Predictive parsing is an improved version of recursive descent parsing that eliminates backtracking. It uses a lookahead symbol to predict which production rule should be applied.

In predictive parsing, the parser decides the correct production rule by examining the next input token. This makes the parsing process faster and more efficient.

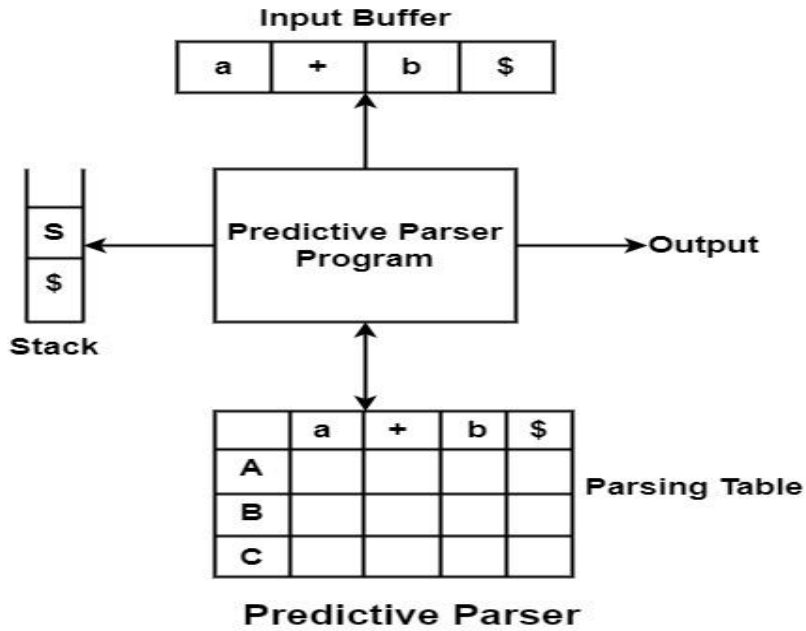


Figure 6.2: Predictive Parsing Process

The working of predictive parsing is shown in **Figure 6.2**, where the parser selects production rules based on the current input symbol

In **Figure 6.2** the parser uses a two-dimensional array or Matrix M [A, a] where A is nonterminal and 'a' is a terminal symbol. And a Stack that contains a combination of grammar symbols with $ on the bottom of the stack. At the start of Parsing, the stack contains the start symbol of Grammar followed by $.

Predictive parsing requires the grammar to be free from left recursion and ambiguity. These conditions ensure that the parser can make decisions using only one lookahead symbol.

6.4 FIRST and FOLLOW

FIRST and FOLLOW are two important concepts used in predictive parsing to construct parsing tables.

The FIRST set of a symbol represents the set of terminals that can appear at the beginning of strings derived from that symbol. The FOLLOW set represents the set of terminals that can appear immediately after a non-terminal in some derivation.

The computed FIRST and FOLLOW sets for a sample grammar are shown in **Table 6.1**.

Table 6.1: FIRST and FOLLOW Sets

Non-terminal	FIRST	FOLLOW
E	{id}	{ $ }
T	{id}	{ +, $ }

In Table 6.1, the FIRST set indicates the possible starting symbols, while the FOLLOW set indicates possible succeeding symbols.

These sets play a crucial role in predictive parsing because they help the parser choose the correct production rule based on the input token.

6.5 LL(1) Parsing Table

An LL(1) parsing table is used in predictive parsing to determine which production rule should be applied for a given non-terminal and input symbol.

The term LL(1) means:

- The input is scanned from Left to right
- The parser produces a Leftmost derivation
- One lookahead symbol is used

The structure of an LL(1) parsing table is shown in **Table 6.2**.

Table 6.2: LL(1) Parsing Table

Non-terminal	id	+	$
E	E → T		
T	T → id		

In Table 6.2, rows represent non-terminals and columns represent input symbols. Each cell contains the production rule to be applied. Some of cells may appear empty. These empty cells indicate that no production rule is applicable for the given combination of non-terminal and input symbol.

This situation occurs when the input symbol does not belong to the FIRST set of any production of that non-terminal, and there is no ε (epsilon) production to use the FOLLOW set.

Therefore, empty entries indicate invalid or unexpected inputs during parsing, and the parser reports a syntax error when such a situation occurs.

When the parser reads a token, it looks up the corresponding cell in the table and applies the rule.

The complete parsing process using an LL(1) table is illustrated in **Figure 6.3**, where the parser uses stack and input to guide parsing decisions.

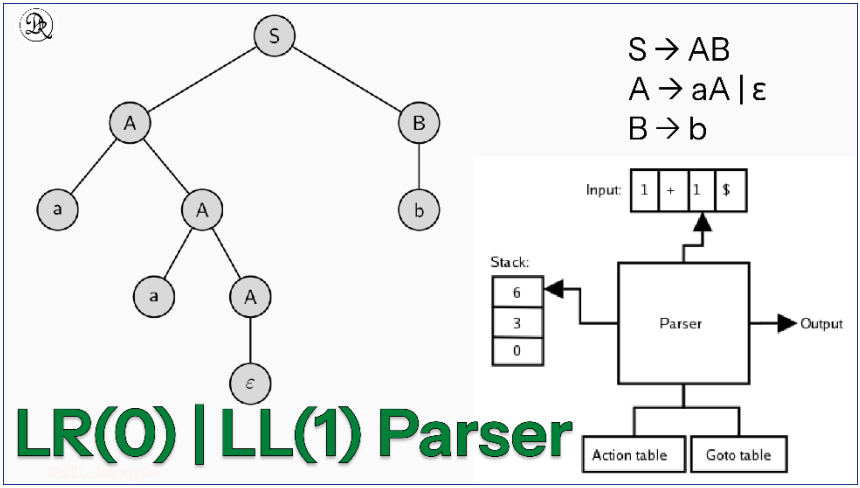


Figure 6.3: LL(1) Parsing Process Using Stack

As shown in Figure 6.3, the parser uses a stack and input buffer to process tokens and apply grammar rules.

EXERCISE

1. Given the grammar:

```
S → A B
A → a
B → b
```

Trace recursive descent parsing for input ab.

2. Construct FIRST and FOLLOW sets for:

```
E → E + T | T
T → T * F | F
F → (E) | id
```

3. Build LL(1) parsing table for a given grammar.
4. Explain left recursion removal using example.
5. Convert a grammar into LL(1) form.
6. Explain predictive parsing with stack operations.
7. Show parsing steps for input a+b.
8. Explain how FIRST and FOLLOW sets are used.
9. Design recursive descent parser for arithmetic expressions.
10. Explain advantages and limitations of top-down parsing.

CHAPTER 7

BOTTOM-UP PARSING

7.1 Introduction to Bottom-Up Parsing

Bottom-up parsing is a technique in syntax analysis where the parse tree is constructed starting from the input string and moving upward toward the start symbol. In contrast to top-down parsing, which begins with the start symbol, bottom-up parsing attempts to reduce the input string step by step until it becomes the start symbol.

The main idea of bottom-up parsing is to repeatedly identify substrings of the input that match the right-hand side of production rules and replace them with the corresponding non-terminal symbols. This process is called reduction.

Consider the following expression:

```
x * y + z
```

The parser starts with the input and gradually reduces parts of the string according to grammar rules. For example, x and y may first be reduced to identifiers, then x * y is reduced to an expression, and finally the entire expression is reduced to the start symbol.

Classification of Bottom-Up Parsers

Bottom-up parsers can be classified based on how they perform reductions and construct parsing tables. The most common types include shift-reduce parsers and LR parsers.

A **shift-reduce parser** is the simplest form of bottom-up parser. It uses a stack to store symbols and performs two main operations: shifting input symbols onto the stack and reducing them using

grammar rules. This parser works by identifying handles and reducing them step by step.

More advanced bottom-up parsers belong to the **LR family**, where "L" stands for left-to-right scanning of input and "R" represents rightmost derivation in reverse. LR parsers are more powerful and can handle a wider range of grammars without ambiguity.

Within LR parsers, there are three important types. The **SLR (Simple LR) parser** uses FOLLOW sets to construct parsing tables and is relatively easy to implement. The **CLR (Canonical LR) parser** is more powerful because it uses complete LR items, but it requires larger parsing tables. The **LALR (Look-Ahead LR) parser** is a compromise between SLR and CLR, combining efficiency with reduced table size, and is widely used in practical compiler construction.

This classification is conceptually illustrated in **Figure 7.1**, where different types of bottom-up parsers are organized hierarchically.

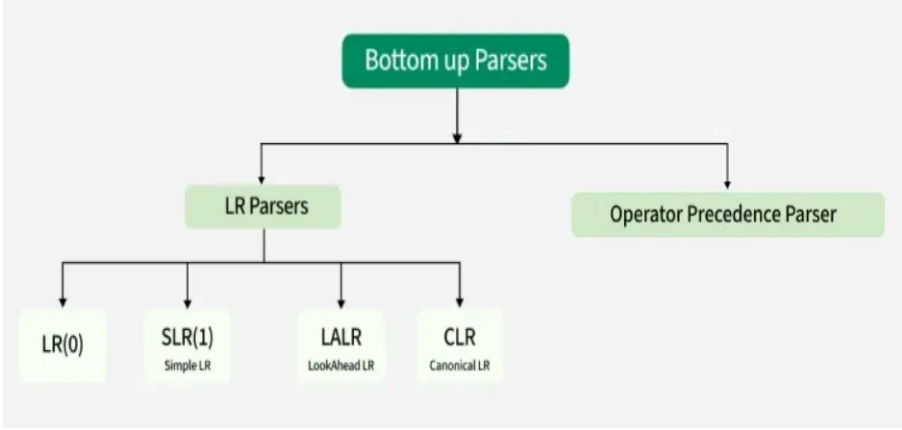


Figure 7.1: Classification of Bottom-Up Parsers

As shown in Figure 7.1, bottom-up parsing techniques range from simple shift-reduce parsing to more powerful LR-based methods used in modern compilers.

7.2 Shift-Reduce Parsing

Shift-reduce parsing is the most common form of bottom-up parsing. It uses a stack to store symbols and processes the input from left to right.

The parsing process consists of two main operations:

- **Shift:** The next input symbol is pushed onto the stack
- **Reduce:** A sequence of symbols on the stack is replaced by a non-terminal

The working of shift-reduce parsing can be understood using the steps shown in **Table 7.1**, where the input string is processed step by step.

Table 7.1: Shift-Reduce Parsing Steps

Stack	Input	Action
$	x*y+z$	Shift
$x	*y+z$	Shift
$x*	y+z$	Shift
$x*y	+z$	Reduce
$E	+z$	Shift
$E+	z$	Shift
$E+z	$	Reduce
$E	$	Accept

In Table 7.1, the parser alternates between shifting symbols onto the stack and reducing them according to grammar rules.

The logic behind this process is represented in **Program 7.1**, where a simple shift-reduce algorithm is described using pseudocode.

Program 7.1: Pseudocode for Shift-Reduce Parsing

```
initialize stack with $
repeat
    if reduction is possible
        perform reduce operation
    else
        shift next input symbol
until input is empty and stack contains start symbol
```

Program 7.1 shows that the parser continuously decides whether to shift or reduce based on the current stack contents.

7.3 Handle Concept

A handle is a substring of a right-sentential form that matches the right-hand side of a production rule and can be reduced to a non-terminal.

In bottom-up parsing, identifying the correct handle is crucial because it determines which part of the string should be reduced at each step.

Consider the expression:

```
id * id
```

During parsing, the substring id * id forms a handle because it matches the production rule for multiplication. This substring is reduced first before handling subtraction.

The identification of handles during parsing is illustrated in **Figure 7.2**, where specific parts of the input string are selected for reduction.

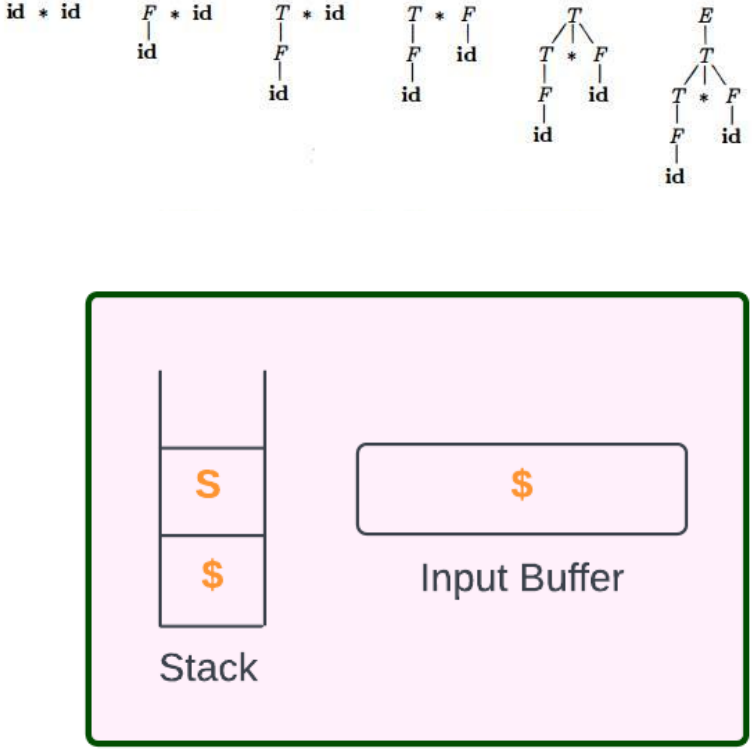


Figure 7.2: Identification of Handle in Bottom-Up Parsing

As shown in Figure 7.2, handles are reduced in a specific order to ensure correct parsing according to operator precedence.

7.4 LR Parsing Overview

LR parsing is a powerful bottom-up parsing technique that can handle a large class of grammars. The name LR stands for:

- **L:** Left-to-right scanning of input
- **R:** Rightmost derivation in reverse

LR parsers use a stack and a parsing table to decide whether to shift or reduce. The parsing decisions are based on the current state and the next input symbol.

The structure of an LR parser is shown in **Figure 7.3**, where the parser interacts with the input buffer, stack, and parsing table.

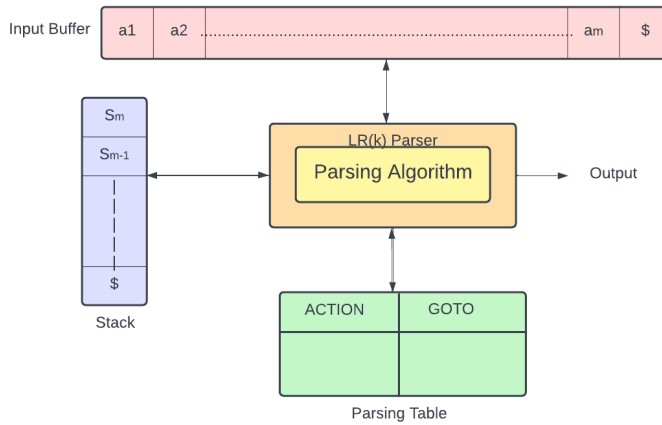


Figure 7.3: Structure of an LR Parser

In Figure 7.3, the parser uses two tables: the action table and the go to table. These tables guide the parsing process by indicating whether to shift, reduce, accept, or report an error.

LR parsing is preferred in practical compiler design because it is more powerful and can handle complex grammars efficiently.

7.5 SLR, LALR, CLR Parsers

LR parsers are classified into different types based on their complexity and power. The three main types are SLR, LALR, and CLR parsers.

SLR: SLR or Simple LR parser is the most basic type of LR parser. It uses the FOLLOW set to decide when a reduction should be applied. Because it uses limited information, it is easy to construct but sometimes fails to handle more complex grammars.

For example,

consider a grammar where both $E \rightarrow E + T$ and $T \rightarrow id$ exist.

When parsing an expression like a + b, the SLR parser may face difficulty deciding whether to reduce or shift if the FOLLOW sets overlap. This shows that SLR is simple but not very powerful.

LALR: LALR or Look-Ahead LR parser is an improved version of SLR. It combines similar LR states to reduce the size of the parsing table while still using lookahead information to make better decisions. This makes it more powerful than SLR but more efficient than full LR parsing.

For instance, when parsing an expression like p * q + r, the LALR parser can correctly decide that multiplication should be reduced before addition because it uses more precise lookahead information. That is why LALR parsers are widely used in real compiler tools like YACC.

CLR: CLR or Canonical LR parser, also known as LR(1) parser, is the most powerful type among the three. It uses complete lookahead information for every parsing decision, which allows it to handle almost all context-free grammars. However, this power comes at the cost of very large parsing tables.

For example, when parsing an expression like x - y * z, the CLR parser always makes the correct decision because it considers both the current state and the next input symbol with full precision. This ensures accurate parsing but requires more memory and complexity.

The comparison among these parsers is shown in **Table 7.2**, where their characteristics are summarized.

Table 7.2: Comparison of LR Parsers

Parser Type	Description	Complexity
SLR	Uses FOLLOW sets for decisions	Simple
LALR	Combines similar states	Moderate
CLR	Uses full LR(1) items	Complex

In Table 7.2, SLR parsers are the simplest but less powerful, while CLR parsers are the most powerful but require large parsing tables. LALR parsers provide a balance between power and efficiency and are commonly used in real compiler tools.

EXERCISE

1. Perform shift-reduce parsing for:

```
x * y + z
```

2. Identify handles in:

```
a + b * c
```

3. Explain shift-reduce conflict using example.
4. Construct parsing steps for bottom-up parsing.
5. Explain LR parsing process with diagram.
6. Compare SLR, LALR, and CLR parsers.
7. Construct parsing table for a simple grammar.
8. Explain reduce-reduce conflict.
9. Trace stack operations in shift-reduce parsing.
10. Explain advantages of bottom-up parsing.

Part IV | Semantic Analysis

CHAPTER 8

SEMANTIC ANALYSIS

8.1 Introduction to Semantic Analysis

Semantic analysis is the third phase of a compiler. After syntax analysis verifies that a program is structurally correct, semantic analysis ensures that the program is logically meaningful.

In simple terms, syntax analysis checks whether the program is written correctly, while semantic analysis checks whether the program makes sense.

For example, consider the statement:

```
int a;

a = "hello";
```

This statement is syntactically correct because it follows the grammar rules. However, it is semantically incorrect because a string value is assigned to an integer variable.

The working of semantic analysis is illustrated in **Figure 8.1**, where the parse tree generated by syntax analysis is checked for meaning and correctness.

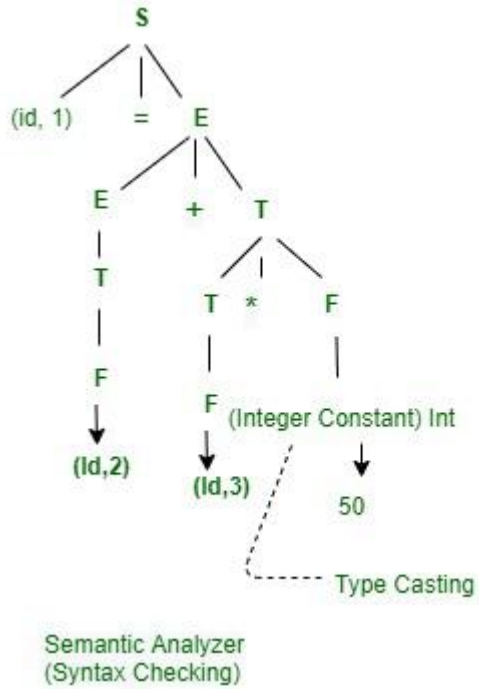


Figure 8.1: Working of Semantic Analysis Phase

As shown in Figure 8.1, the semantic analyzer verifies the meaning of the program using information such as types, scopes, and symbol tables.

8.2 Type Checking

Type checking is one of the main tasks of semantic analysis. It ensures that operations are performed on compatible data types.

For example, consider the following expression:

```
float x;

int y;

x = y + 2;
```

This expression is valid because the integer value can be converted into a float. However, the following statement is invalid:

```
int x;

x = 3.5;
```

Here, a floating-point value is assigned to an integer variable without proper conversion.

The rules used for type checking are summarized in **Table 8.1**, where valid and invalid operations are shown.

Table 8.1: Type Checking Examples

Expression	Result
int + int	Valid
int + float	Valid (with conversion)
int + string	Invalid

In Table 8.1, valid operations involve compatible types, while invalid operations lead to semantic errors.

8.3 Symbol Table Usage

The symbol table is a data structure used by the compiler to store information about identifiers such as variables, functions, and constants.

During semantic analysis, the symbol table is used to check whether variables are declared before use and whether they are used correctly.

For example:

```
int a;

a = 10;

b = 5;
```

In this case, variable b is not declared, which results in a semantic error.

The structure of a symbol table is shown in **Table 8.2**, where information about variables is stored.

Table 8.2: Example Symbol Table

Name	Type	Scope
a	int	local
x	float	global

As shown in Table 8.2, the symbol table stores attributes such as name, type, and scope, which are essential for semantic analysis.

8.4 Scope Management

Scope refers to the region of a program where a variable is accessible. Semantic analysis ensures that variables are used within their defined scope.

Consider the following example:

```
int a = 5;

{

    int b = 10; }
```

Here, variable b is only accessible within the block. If it is accessed outside the block, it results in an error.

The concept of scope is illustrated in **Figure 8.2**, where different regions of a program are shown.

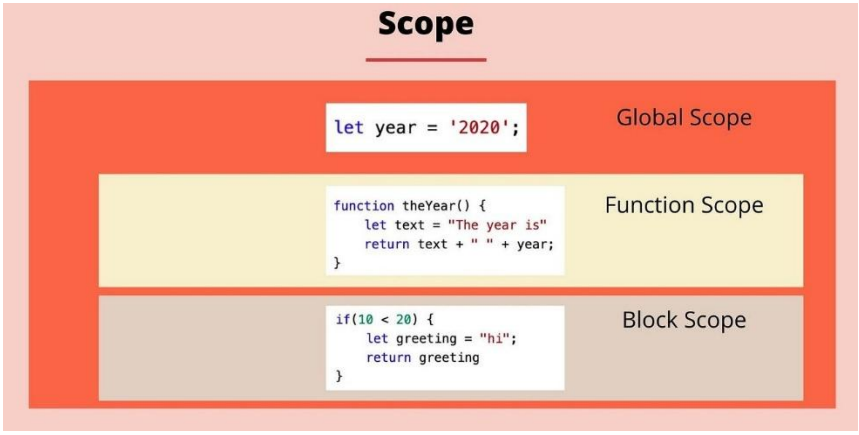


Figure 8.2: Scope Management in a Program

As shown in Figure 8.2, variables defined inside a block are not accessible outside that block.

8.5 Type Conversion

Type conversion is the process of converting one data type into another. It can be implicit (automatic) or explicit (manual).

Consider the example:

```
int a = 5;

float b = 2.5;

float c = a + b;
```

Here, the integer value of a is automatically converted to float before addition. This is known as implicit conversion.

Explicit conversion is performed by the programmer:

```
int x = (int)3.8;
```

In this case, the floating-point value is explicitly converted into an integer.

The concept of type conversion is summarized in **Table 8.3**.

Table 8.3: Type Conversion Examples

Conversion Type	Example
Implicit	int → float
Explicit	(int) 3.8

As shown in Table 8.3, type conversion helps maintain compatibility between different data types.

8.6 Semantic Errors

Semantic errors occur when a program is syntactically correct but logically incorrect.

Examples include:

```
int x;

x = y + 5;
```

Here, variable y is not declared, resulting in a semantic error.

Another example is:

```
int a;

a = "text";
```

This involves incompatible data types.

The types of semantic errors are illustrated in **Figure 8.3**, where different error categories are shown.

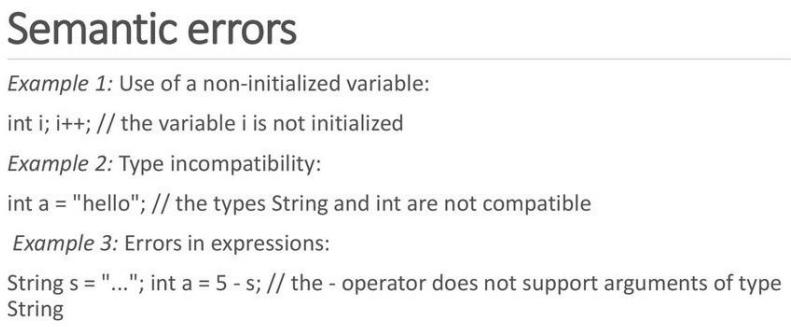

Semantic errors
Example 1: Use of a non-initialized variable:
int i; i++; // the variable i is not initialized
Example 2: Type incompatibility:
int a = "hello"; // the types String and int are not compatible
Example 3: Errors in expressions:
String s = "..."; int a = 5 - s; // the - operator does not support arguments of type String

Figure 8.3: Types of Semantic Errors

As shown in Figure 8.3, semantic errors include undeclared variables, type mismatches, and incorrect operations.

8.7 Syntax Directed Translation

Syntax-directed translation is a technique used to attach semantic rules to grammar productions. It helps in generating intermediate code and performing semantic checks during parsing.

Each production rule is associated with actions that define how values are computed.

The working of syntax-directed translation is illustrated in **Figure 8.4**, where grammar rules are combined with semantic actions.

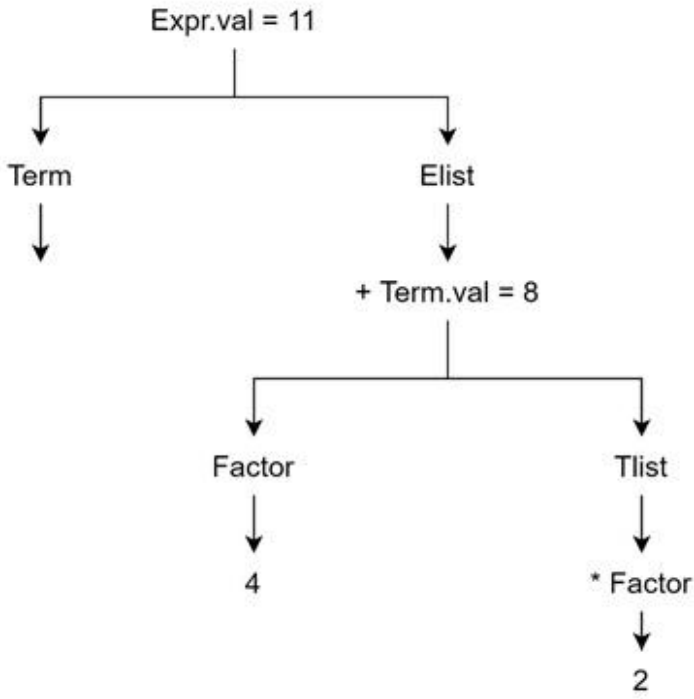


Figure 8.4: Syntax Directed Translation Process

As shown in Figure 8.4, semantic actions are executed during parsing to generate intermediate representations.

For example, consider the expression:

```
result = a + b;
```

The parser not only checks syntax but also generates intermediate code using semantic rules.

EXERCISE

1. Analyze the following code:

```
int x;
x = "text";
```

Explain the error and phase.

2. Perform type checking for:

```
float a;
int b;
a = b + 2;
```

3. Explain scope rules using nested blocks.
4. Construct symbol table for:

```
int a;
float b;
```

5. Explain semantic rules in syntax-directed translation.
6. Identify semantic errors in a program.
7. Explain type conversion with example.
8. Describe attribute grammar with example.
9. Show semantic analysis of an expression.
10. Explain role of semantic analyzer in compiler.

Part V | Intermediate Code Generation

CHAPTER 9

INTERMEDIATE CODE

9.1 Introduction to Intermediate Representation

Intermediate representation (IR) is a form of program representation used between the source code and the target machine code. After semantic analysis, the compiler converts the program into an intermediate form that is easier to analyze and optimize.

The main purpose of intermediate representation is to make the compiler independent of machine architecture. Instead of directly generating machine code for different systems, the compiler first produces an intermediate code and then translates it into target code.

For example, consider the expression:

```
total = a + b * c;
```

Instead of directly converting this into machine instructions, the compiler first converts it into a simpler intermediate form such as:

```
t1 = b * c

t2 = a + t1

total = t2
```

In intermediate code, **t1, t2** are temporary variables used to store intermediate results. These temporary variables help break complex expressions into simple steps, making it easier for the compiler to process and generate code. The overall transformation where the intermediate representation acts as a bridge between analysis and code generation phases.

9.2 Syntax Trees

A syntax tree is a simplified form of parse tree that represents the structure of a program in a hierarchical manner. Unlike parse trees, syntax trees remove unnecessary grammar details and focus only on essential operations.

Each internal node represents an operator, and each leaf node represents an operand.

Consider the expression:

```
x = p - q * r;
```

The corresponding syntax tree is shown in **Figure 9.1**, where multiplication is performed before subtraction.

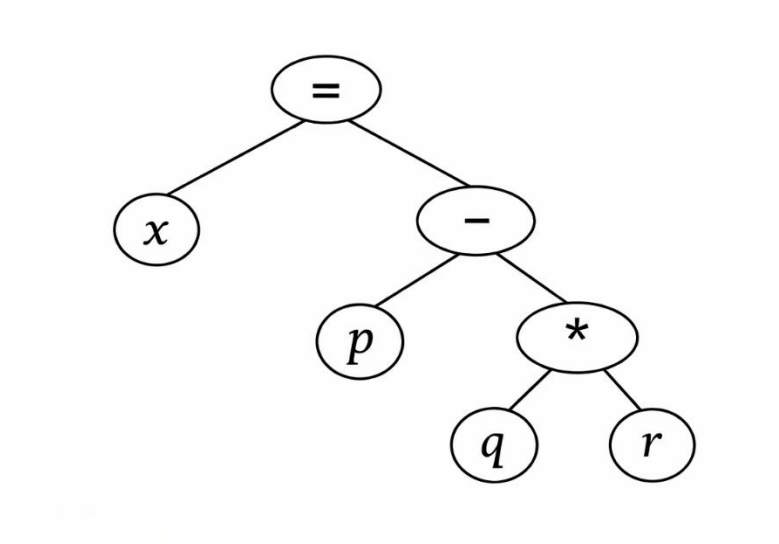


Figure 9.1: Syntax Tree for Expression x = p - q * r

As shown in Figure 9.1, the tree clearly represents operator precedence and structure without including unnecessary grammar symbols.

9.3 Directed Acyclic Graph (DAG)

A Directed Acyclic Graph (DAG) is an advanced form of syntax tree used to represent expressions more efficiently. It eliminates repeated computations by sharing common subexpressions.

Unlike syntax trees, DAG allows multiple parents for a node, which helps avoid duplicate calculations.

Consider the expression:

```
(a + b) + (e + (c - d))
```

Instead of computing a + b twice, a DAG represents it once and reuses the result. This structure is illustrated in **Figure 9.2**, where the common subexpression is shared.

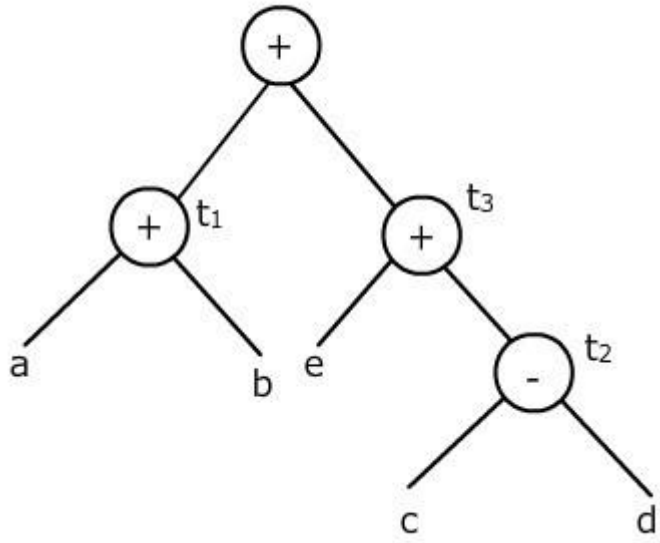


Figure 9.2: DAG Representation with Common Subexpression

As shown in Figure 9.2, DAG helps optimize code by avoiding redundant computations.

9.4 Three Address Code (TAC)

Three Address Code (TAC) is a widely used intermediate representation where each instruction contains at most three operands. It breaks complex expressions into simple steps using temporary variables.

Consider the expression:

```
result = (m + n) * (p - q);
```

The corresponding three address code is:

```
t1 = m + n

t2 = p - q

t3 = t1 * t2

result = t3
```

In TAC, each instruction performs a single operation, making it easier for the compiler to optimize and generate machine code.

The structure of TAC is illustrated in **Figure 9.3**, where complex expressions are broken into smaller operations.

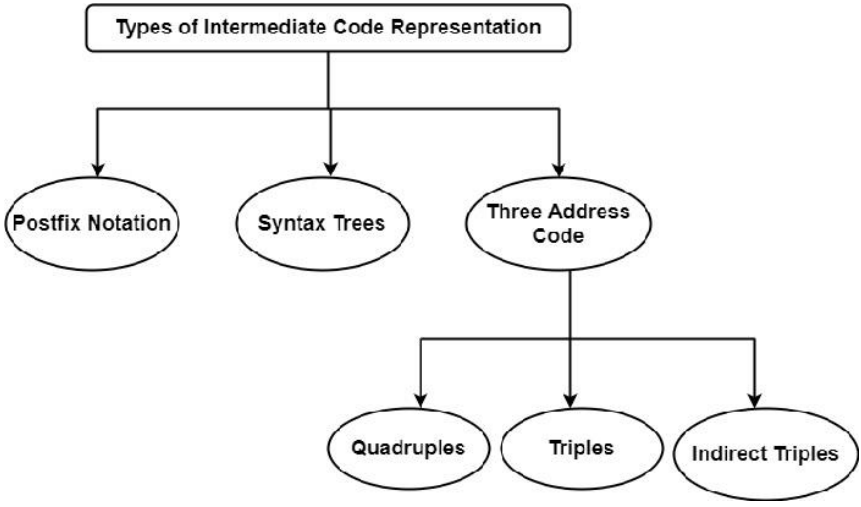


Figure 9.3: Structure of Three Address Code

As shown in Figure 9.3, TAC simplifies expressions by dividing them into sequential steps.

9.5 Quadruples, Triples, Indirect Triples

Three address code can be represented in different forms depending on how the operations and operands are stored.

Quadruples store each instruction in four fields: operator, operand1, operand2, and result. For example, the instruction t1 = m + n is stored as (+, m, n, t1).

Triples eliminate the result field and instead use positions (indices) to refer to intermediate results. For example, the result of one operation can be referred to by its position in the instruction list.

Indirect triples extend triples by using a separate list of pointers to improve code rearrangement and optimization.

The relationship among these representations is illustrated in **Figure 9.4**, where the same expression is represented in three different formats.

		Operator	Operand1	Operand2
(1)	(1)	+	a	b
(2)	(2)	^	c	
(3)	(3)	*	(1)	(2)
(4)	(4)	/	(3)	d
(5)	(5)	=	x	(4)
Indirect Triple Representation of $x = (a + b) * ^ c/d$				

Figure 9.4: Representation of TAC Using Quadruples, Triples, and Indirect Triples

As shown in Figure 9.4, all three methods represent the same operations but differ in how intermediate results are stored and accessed.

EXERCISE

1. Convert expression to TAC:

```
a = b + c * d;
```

2. Construct syntax tree for:

```
x = y - z * w;
```

3. Build DAG for common subexpression elimination.
4. Represent TAC using quadruples.
5. Represent TAC using triples.
6. Explain indirect triples.
7. Compare syntax tree and DAG.
8. Generate intermediate code for:

```
p = q * r + s;
```

9. Explain advantages of intermediate code.
10. Show step-by-step IR generation.

Part VI | Code Optimization

CHAPTER 10

CODE OPTIMIZATION

10.1 Introduction to Optimization

Code optimization is the process of improving intermediate code so that the final generated machine code becomes faster and more efficient. This phase takes place after intermediate code generation and before code generation.

The primary goal of optimization is not to change the meaning of the program but to improve its performance. This may involve reducing execution time, minimizing memory usage, or simplifying computations.

For example, consider the following intermediate code:

```
t1 = 4 * 5

x = t1
```

This can be optimized into:

```
x = 20
```

The transformation avoids unnecessary computation and improves efficiency.

The overall role of optimization in the compiler is illustrated in **Figure 10.1**, where intermediate code is refined before final code generation.

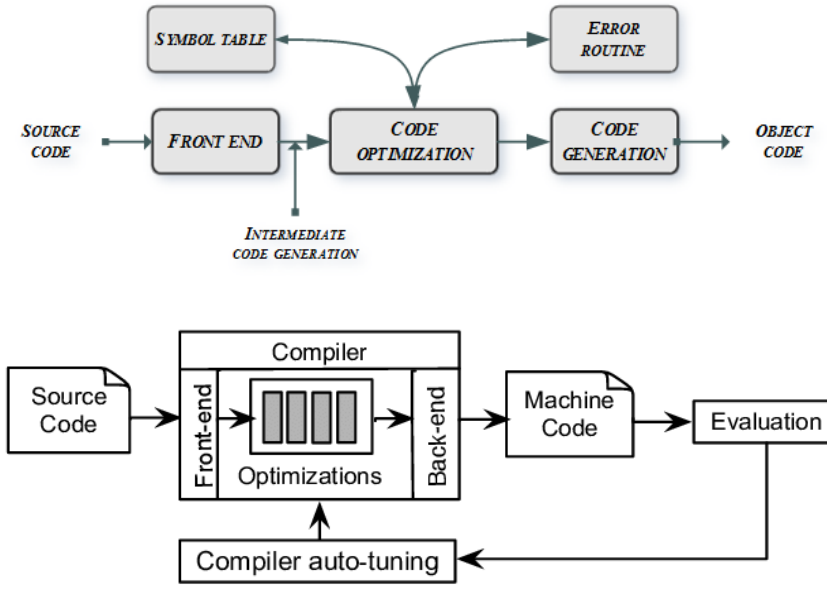


Figure 10.1: Position of Code Optimization in Compilation

As shown in Figure 10.1, optimization improves the intermediate code before it is converted into machine instructions.

10.2 Goals of Optimization

The purpose of optimization is to improve program performance without altering its output. The optimization process aims to produce code that executes faster and uses fewer resources.

For instance, a program may contain repeated calculations or unnecessary instructions that increase execution time. Optimization removes these inefficiencies while maintaining correctness.

Consider the example:

```
y = (2 * 3) + z
```

This can be optimized into:

```
y = 6 + z
```

This reduces computation and improves performance.

The key idea is that optimization enhances efficiency while preserving the original semantics of the program.

10.3 Types of Optimizations

Code optimization can be broadly classified into two categories based on the stage at which it is applied.

Machine-independent optimization is performed on intermediate code and does not depend on the target machine architecture. Techniques such as constant folding and dead code elimination fall under this category.

Machine-dependent optimization is applied during code generation and focuses on improving performance based on hardware characteristics such as registers and instruction sets.

The classification of optimization techniques is illustrated in **Figure 10.2**, where both types are shown in relation to compiler phases.

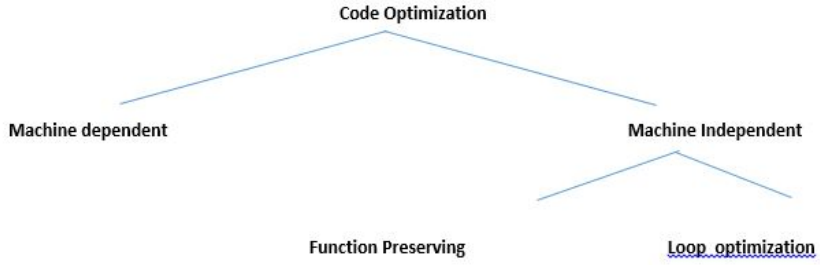


Figure 10.2: Types of Code Optimization

As shown in Figure 10.2, machine-independent optimization improves the structure of intermediate code, while machine-dependent optimization focuses on efficient execution on specific hardware.

10.4 Basic Blocks

A basic block is a sequence of consecutive statements in which control enters at the beginning and exits at the end without any branching in between.

In other words, once execution starts in a basic block, all statements within it are executed sequentially without interruption.

Consider the following code:

```
t1 = a + b

t2 = t1 * c

t3 = t2 - d
```

This forms a single basic block because there are no jumps or branches within the sequence.

The structure of a basic block is shown in **Figure 10.3**, where instructions are executed linearly.

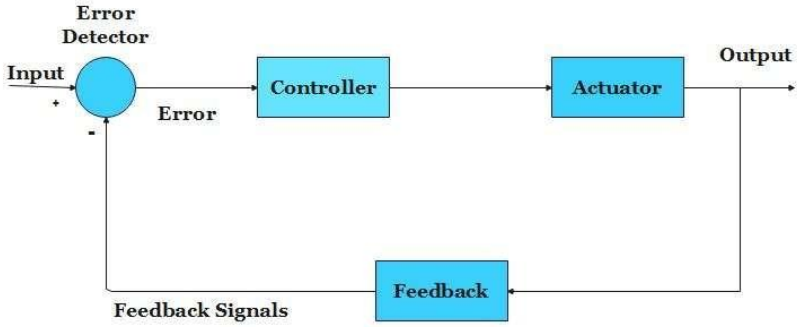


Figure 10.3: Basic Block Representation

As illustrated in Figure 10.3, optimization techniques are often applied within basic blocks to improve performance.

10.5 Optimization Techniques

Code optimization techniques are applied to intermediate code in order to improve efficiency without changing the meaning of the program. These techniques mainly focus on reducing unnecessary computations, eliminating redundancy, and improving execution speed. In practical compilers, multiple optimization techniques are often applied together to achieve better performance.

10.5.1 Constant Folding

Constant folding is a technique where expressions involving constant values are evaluated at compile time rather than at runtime. This reduces the number of operations that need to be performed during execution.

Consider the following intermediate code:

```
t1 = 6 * 4

y = t1 + k
```

Here, the multiplication 6 * 4 does not depend on any variable, so it can be computed during compilation. After applying constant folding, the optimized code becomes:

```
y = 24 + k
```

This eliminates one runtime operation. In large programs, such simplifications significantly reduce execution time.

Constant folding is particularly useful when expressions contain multiple constants, such as:

```
z = (3 * 5) + (8 - 2)
```

which can be directly simplified to:

```
z = 15 + 6

z = 21
```

Thus, constant folding reduces computational overhead and improves performance.

10.5.2 Dead Code Elimination

Dead code elimination removes instructions that do not affect the final output of the program. Such code may arise due to redundant assignments or unused variables.

Consider the following sequence:

```
x = 12

x = 25

print(x)
```

The first assignment x = 12 is never used because it is overwritten before any use. Therefore, it is considered dead code. The optimized version becomes:

```
x = 25

print(x)
```

Another example involves unused variables:

```
a = 5

b = a + 2
```

If b is never used later in the program, the entire computation b = a + 2 can be removed.

Dead code elimination reduces program size and improves execution efficiency by removing unnecessary instructions.

10.5.3 Common Subexpression Elimination

Common subexpression elimination avoids repeated computation of the same expression when its value has not changed.

Consider the following code:

```
t1 = u + v

t2 = u + v

result = t1 - t2
```

The expression u + v is computed twice, even though its value remains unchanged. This redundancy can be removed by computing it once:

```
t1 = u + v

t2 = t1

result = t1 - t2
```

Now the computation is performed only once, reducing execution time.

This technique becomes even more important in large programs where the same expression appears multiple times. Eliminating repeated calculations significantly improves efficiency.

10.5.4 Copy Propagation

Copy propagation replaces occurrences of a variable that simply copies another variable. This reduces unnecessary assignments and simplifies expressions.

Consider the code:

```
m = n
```

```
p = m * 3
```

Here, m is just a copy of n, so its use can be replaced directly:

```
p = n * 3
```

After this transformation, the assignment m = n may become unnecessary and can be removed by dead code elimination.

Copy propagation works effectively with other optimization techniques. By reducing intermediate variables, it simplifies the code and improves readability and performance.

10.5.5 Loop Optimization

Loop optimization focuses on improving the performance of loops, since loops often execute many times and contribute significantly to total execution time.

One of the most important techniques in loop optimization is **loop-invariant code motion**, where expressions that do not change within the loop are moved outside.

Consider the following loop:

```
for(i = 0; i < n; i++)

    sum = sum + (a * b);
```

The expression a * b does not change during loop execution. Therefore, computing it repeatedly is inefficient. The optimized version becomes:

```
t1 = a * b

for(i = 0; i < n; i++)

    sum = sum + t1;
```

This reduces repeated computation inside the loop, leading to better performance.

The working of loop optimization is illustrated in **Figure 10.4**, where invariant expressions are moved outside the loop body.

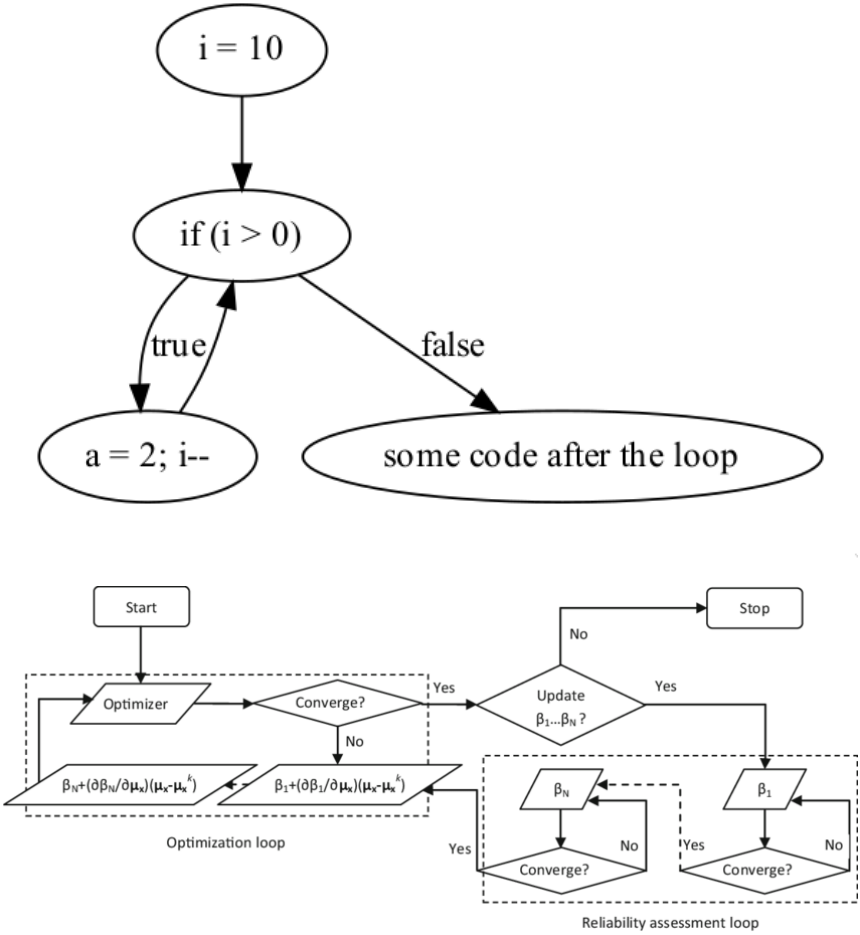


Figure 10.4: Loop Optimization (Loop-Invariant Code Motion)

As shown in Figure 10.4, expressions that remain constant during loop execution are computed once, reducing the number of operations inside the loop.

EXERCISE

1. Identify all the **basic blocks** in given code and explain how they are formed.

```
a = b + c

d = a - e

if d > 0 goto L1

f = d * 2

goto L2

L1: f = d + 2

L2: g = f + 1
```

2. Perform dead code elimination.
3. Apply constant folding:

```
x = 2 + 3;
```

4. Perform common subexpression elimination.
5. Explain loop optimization.
6. Construct control flow graph.
7. Perform data flow analysis.
8. Explain optimization techniques.
9. Optimize given code.
10. Explain global vs local optimization.

Part VII | Code Generation

CHAPTER 11

CODE GENERATION

11.1 Introduction to Code Generation

Code generation is the final phase of a compiler, where the intermediate representation of a program is converted into target machine code or assembly code. This phase directly determines how efficiently the program will run on a machine.

After intermediate code is produced and optimized, the code generator translates it into instructions that can be executed by the processor. The quality of generated code is important because it affects both execution speed and memory usage.

For example, consider the intermediate code:

```
t1 = x + y

z = t1
```

The code generator converts this into machine-like instructions such as:

```
MOV R1, x

ADD R1, y

MOV z, R1
```

The overall process of code generation is illustrated in **Figure 11.1**, where intermediate code is translated into target code using machine-specific instructions.

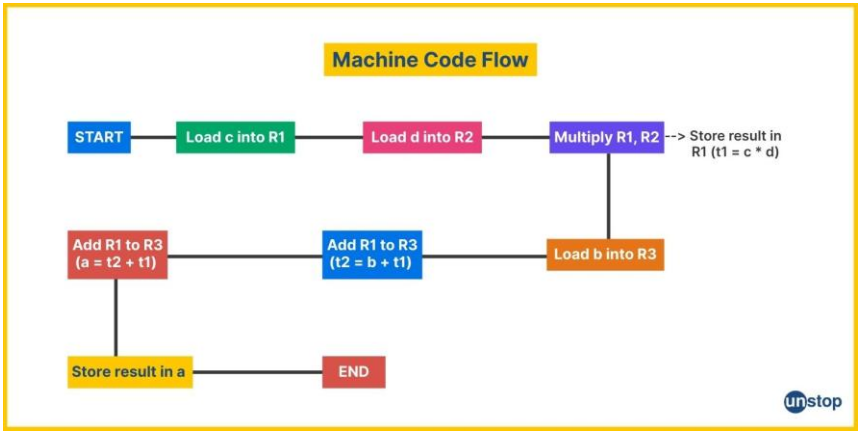

Machine Code Flow
START
Load c into R1
Load d into R2
Multiply R1, R2
--> Store result in R1 (t1 = c * d)
Add R1 to R3 (a = t2 + t1)
Add R1 to R3 (t2 = b + t1)
Load b into R3
Store result in a
END
unstop

Figure 11.1: Code Generation Process

As shown in Figure 11.1, the code generator forms the final stage of compilation, producing executable instructions.

11.2 Role of Code Generator

The code generator performs several important tasks in transforming intermediate code into target code. It ensures that the generated instructions are correct and efficient.

The main responsibility of the code generator is to map intermediate operations into machine instructions. It also manages the use of CPU registers, decides where variables should be stored, and ensures proper use of memory.

Consider the intermediate code:

```
t1 = a * b

t2 = t1 + c
```

The code generator must decide how to translate these operations into instructions using available registers and memory locations.

The internal working of the code generator is represented in **Figure 11.2**, where different tasks such as instruction selection and register allocation are shown.

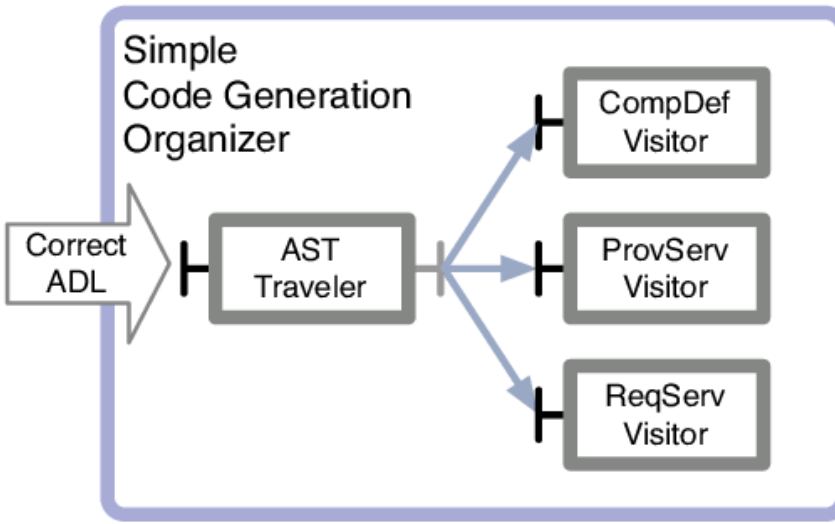


Figure 11.2: Role of Code Generator in Compiler

As illustrated in Figure 11.2, the code generator is responsible for converting intermediate operations into efficient machine-level instructions.

11.3 Input and Output of Code Generator

The input to the code generator is typically the intermediate representation produced by earlier phases of the compiler. This may include three address code, syntax trees, or DAG representations.

The output of the code generator is the target program, which may be in the form of assembly code or machine code.

For example, consider the intermediate code:

```
t1 = p - q
```

```
r = t1 * s
```

The generated output may look like:

```
MOV R1, p

SUB R1, q

MOV R2, s

MUL R1, R2

MOV r, R1
```

The code generator acts as a bridge between intermediate code and executable code.

11.4 Register Allocation

Register allocation is the process of assigning variables and temporary values to a limited number of CPU registers. Efficient register usage is essential for producing fast code.

Since registers are limited, the code generator must decide which variables should remain in registers and which should be stored in memory.

Consider the example:

```
x = a + b;

y = x + c;
```

If registers are used efficiently, the intermediate result x can remain in a register, avoiding unnecessary memory access.

The concept of register allocation is illustrated in **Figure 11.3**, where variables are mapped to available registers.

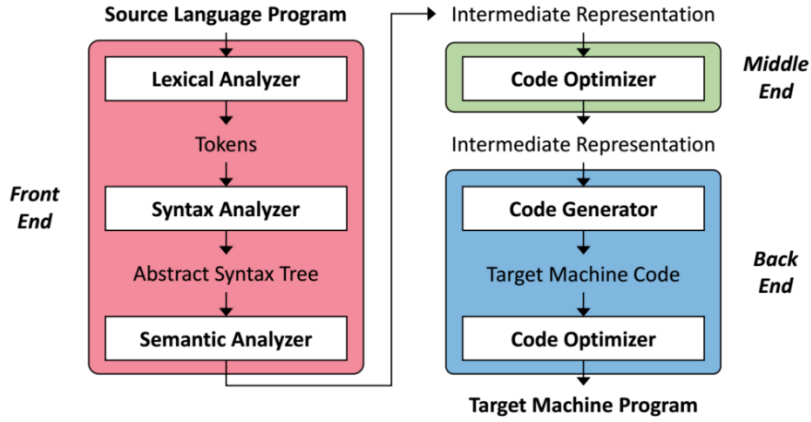


Figure 11.3: Register Allocation Process

As shown in Figure 11.3, efficient allocation reduces memory operations and improves performance.

11.5 Instruction Selection

Instruction selection is the process of choosing appropriate machine instructions to implement operations in the intermediate code. Different machines may have different instruction sets, so the code generator must select instructions carefully.

For example, the expression:

```
k = m + n;
```

can be implemented using different instructions depending on the machine architecture. A simple version may be:

```
MOV R1, m

ADD R1, n

MOV k, R1
```

However, some machines may support combined instructions that perform operations more efficiently.

11.6 Code Generation Examples

To understand the complete process, consider the following expression:

```
result = (u + v) * (w - x);
```

First, the intermediate code is generated:

```
t1 = u + v

t2 = w - x

t3 = t1 * t2

result = t3
```

Then, the code generator produces machine-like instructions:

```
MOV R1, u

ADD R1, v

MOV R2, w

SUB R2, x

MUL R1, R2

MOV result, R1
```

This step-by-step transformation is illustrated in **Figure 11.4**, where the entire code generation process is shown from intermediate code to final instructions.

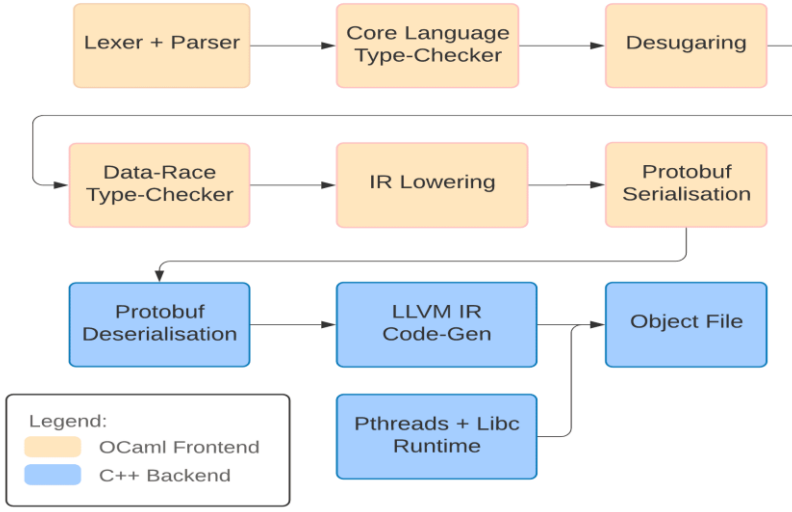


Figure 11.4: Example of Code Generation Process

As shown in Figure 11.4, complex expressions are broken into simpler steps and then converted into executable instructions.

EXERCISE

1. Convert TAC to assembly code.
2. Perform register allocation.
3. Explain instruction selection.
4. Generate machine code for:

```
x = y + z;
```

5. Explain memory usage in code generation.
6. Trace code generation steps.
7. Optimize register usage.
8. Explain code generator design.
9. Generate code for arithmetic expressions.
10. Explain target machine considerations.

Part VIII | Compiler Construction Project

CHAPTER 12

MINI COMPILER DESIGN

12.1 Introduction to Mini Compiler

A mini compiler is a simplified version of a full compiler that demonstrates the core phases of compilation in a practical way. It is designed mainly for learning purposes and academic projects, where students can understand how a compiler works internally.

Unlike a complete compiler, a mini compiler usually supports a limited set of language features such as arithmetic expressions, simple assignments, and basic operators. However, it still follows the same fundamental structure as a real compiler.

For example, a mini compiler may process a statement like:

```
result = a + b * c;
```

and convert it step by step into intermediate code and finally into machine-like instructions.

The overall idea of a mini compiler is illustrated in **Figure 12.1**, where the input program passes through different phases to produce the final output.

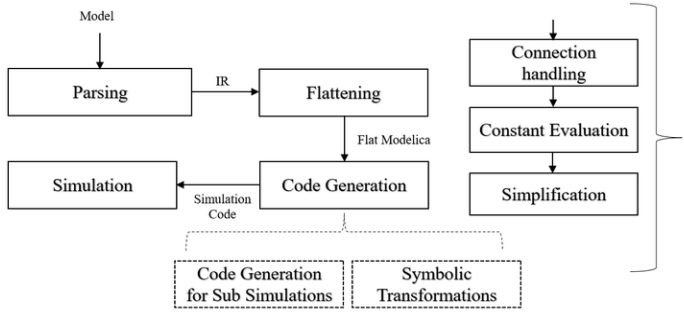


Figure 12.1: Overview of Mini Compiler

As shown in Figure 12.1, even a small compiler follows the same pipeline as a full-scale compiler.

12.2 Compiler Architecture

The architecture of a mini compiler is organized into a set of well-defined modules, where each module performs a specific function in the overall compilation process. These modules work together in a coordinated manner to transform a high-level source program into an executable form.

The compilation process begins with the **lexical analyzer**, which reads the source code character by character and converts it into a sequence of tokens. These tokens represent meaningful units such as keywords, identifiers, operators, and constants.

Next, the **syntax analyzer** (parser) receives the tokens and checks whether they follow the grammatical rules of the programming language. It constructs a structured representation, usually in the form of a parse tree or syntax tree.

After syntactic analysis, the **semantic analyzer** verifies the meaning of the program. It performs tasks such as type checking, scope resolution, and consistency checking to ensure that the program is logically correct.

The validated structure is then passed to the **intermediate code generator**, which converts it into an intermediate representation (IR). This representation is machine-independent and simplifies further processing.

In some compiler designs, this intermediate code is further improved by an **optimization phase**, which enhances performance by eliminating unnecessary computations and improving efficiency.

Finally, the **code generator** translates the optimized intermediate representation into target code, which may be assembly language or machine code, depending on the system.

Each module takes input from the previous stage and passes its output to the next stage, forming a pipeline structure. This modular design improves maintainability, scalability, and clarity of the compiler.

Explanation of Flow

- **Source Code → Lexical Analyzer**
  Converts characters into tokens
- **Tokens → Syntax Analyzer**
  Checks structure and builds parse tree
- **Syntax Tree → Semantic Analyzer**
  Validates meaning and types
- **Validated Data → Intermediate Code Generator**
  Produces machine-independent code
- **Intermediate Code → Code Generator**
  Produces final executable code

Key Features of This Architecture

- **Modular Design:** Each phase performs a specific task
- **Pipeline Processing:** Output of one phase becomes input to the next
- **Reusability:** Modules can be modified independently
- **Error Detection:** Errors can be detected at different stages

In a mini compiler, some advanced phases such as optimization may be simplified or omitted, but the overall architecture remains similar to that of a full-scale compiler.

12.3 Step-by-Step Compilation Process

The compilation process in a mini compiler can be understood step by step. Each phase transforms the input into a more structured form.

Consider the following input:

```
x = y + 10;
```

First, the lexical analyzer breaks the input into tokens such as identifier, operator, and number. Then, the syntax analyzer checks whether the token sequence matches the grammar rules.

After that, the semantic analyzer ensures that variables are declared and types are compatible. Next, intermediate code is generated in a simplified form, and finally, machine-like instructions are produced.

The flow of this process is illustrated in **Figure 12.2**, where each stage of compilation is shown in sequence.

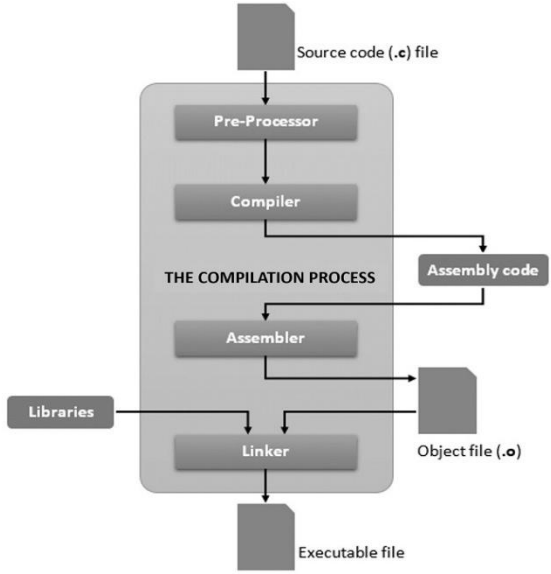


Figure 12.2: Step-by-Step Compilation Process

As shown in Figure 12.2, the program gradually transforms from raw text into executable instructions.

12.4 Grammar Design

Grammar design is a crucial part of building a mini compiler. It defines how valid statements in the language are formed.

For a simple mini compiler, a basic grammar can be defined to handle assignment and arithmetic expressions.

For example:

```
stmt → id = expr

expr → expr + term | term

term → term * factor | factor

factor → id | num
```

This grammar allows expressions involving addition and multiplication, as well as assignment statements.

12.5 Example Walkthrough

To understand the complete working of a mini compiler, consider the following example:

```
z = a + b * d;
```

First, the lexical analyzer identifies tokens such as identifiers and operators. Then, the syntax analyzer constructs a structure based on grammar rules.

After that, semantic analysis ensures that all variables are valid and types are compatible.

Next, intermediate code is generated `t1 = b * d`

```
t2 = a + t1

z = t2
```

Finally, the code generator produces machine-like instructions:

```
MOV R1, b

MUL R1, d

MOV R2, a

ADD R2, R1

MOV z, R2
```

The complete transformation is illustrated in **Figure 12.3**, where each phase contributes to the final output.

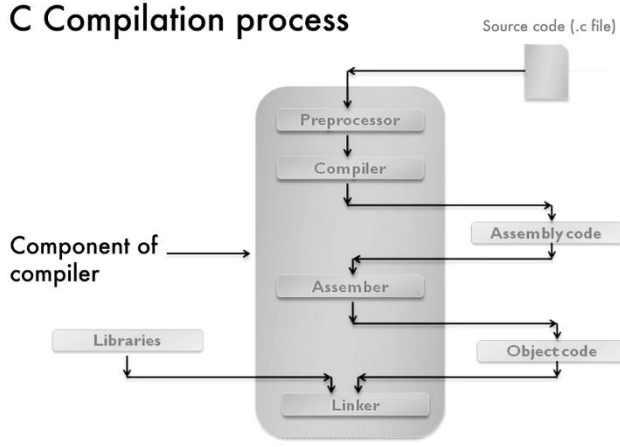


Figure 12.3: Complete Mini Compiler Example

As shown in Figure 12.3, the compiler gradually transforms a simple statement into executable instructions.

EXERCISE

1. Design mini compiler for arithmetic expressions.
2. Explain compilation steps for:

```
x = y + z;
```

3. Design grammar for mini compiler.
4. Explain architecture of mini compiler.
5. Trace execution of mini compiler.
6. Generate tokens, parse tree, and code.
7. Explain implementation challenges.
8. Design scanner and parser.
9. Explain compilation phases in project.
10. Show full example walkthrough.

CHAPTER 13

LEX AND YACC

13.1 Introduction to LEX

LEX is a tool used to automatically generate lexical analyzers. Instead of writing a scanner manually, the programmer defines token patterns using regular expressions, and LEX converts those definitions into a working C program.

The primary function of LEX is to scan the input program and divide it into tokens such as identifiers, keywords, numbers, and operators.

13.2 Structure of LEX Program

A LEX program follows a standard structure divided into three sections. This structure is shown in **Program 13.1**.

Program 13.1: Structure of a LEX Program

```
%{
C declarations
%}
%%
Pattern    Action
%%
User code
```

In Program 13.1, the first section (between %{ and %}) is used for C declarations. This section typically includes header files such as

stdio.h, global variables, and function declarations. These declarations are copied directly into the generated C program.

The second section, located between the two %% symbols, contains pattern-action rules. This is the most important part of the LEX program. Each rule consists of a pattern written using regular expressions and an action written in C. When a pattern matches a part of the input, its corresponding action is executed.

The third section contains user-defined code such as helper functions or the main() function. This code is also included in the final generated program.

A simple rule example is given below:

```
[0-9]+   { printf("NUMBER"); }
```

In this rule, the pattern [0-9]+ represents one or more digits. Therefore, any numeric value such as 5, 27, or 103 will match this pattern. The plus symbol indicates repetition.

The action { printf("NUMBER"); } means that whenever a number is detected, the word "NUMBER" will be printed. For example, if the input is 245, the output will be:

NUMBER

This example demonstrates how LEX connects regular expressions with executable actions.

13.3 LEX Examples

A simple LEX example is shown below:

```
%%

[a-zA-Z]+   { printf("ID\n"); }
```

```
[0-9]+      { printf("NUM\n"); }

"+"         { printf("PLUS\n"); }

%%
```

This program recognizes identifiers, numbers, and the plus operator.

The first rule matches alphabetic strings using [a-zA-Z]+. This pattern represents one or more letters, so inputs like x, abc, or Total are treated as identifiers, and the action prints ID.

The second rule matches numeric values using [0-9]+. When such values are encountered, the program prints NUM.

The third rule matches the symbol +, and the action prints PLUS.

Now consider the input:

```
x + 25
```

The scanner processes the input from left to right. The symbol x matches the identifier pattern and produces ID. The symbol + matches the third rule and produces PLUS. The value 25 matches the numeric pattern and produces NUM.

Thus, the output becomes:

```
ID

PLUS

NUM
```

This token generation process is illustrated in **Figure 13.1**, where each input component is mapped to a token.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Figure 13.1: Token Generation Using LEX

13.4 Introduction to YACC

YACC is a tool used to generate parsers automatically. It takes a grammar specification as input and produces a parser that checks whether a sequence of tokens follows the rules of the language.

While LEX performs lexical analysis, YACC performs syntax analysis. It verifies the structure of tokens and ensures that they form valid expressions.

The working of YACC is illustrated in **Figure 13.2**, where tokens from LEX are processed to build a syntactic structure.

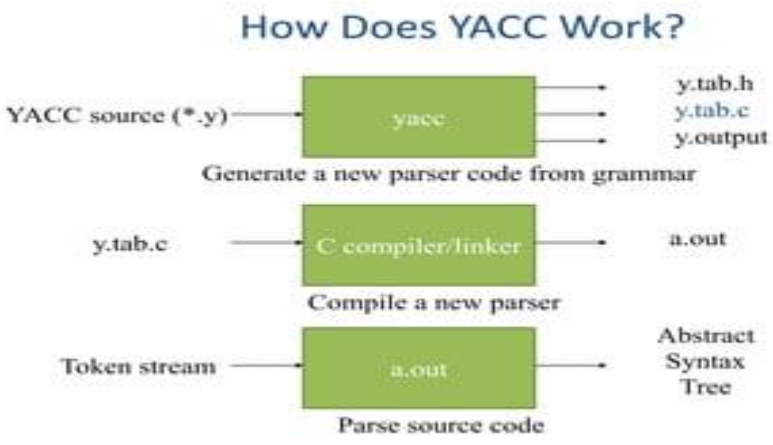


Figure 13.2: Working of YACC Tool

13.5 Structure of YACC Program

A YACC program is also divided into three sections, as shown in **Program 13.2**.

Program 13.2: Structure of a YACC Program

```
%{

C declarations

%}

%%

Grammar rules

%%

User code
```

The first section contains C declarations, which include header files, variables, and function prototypes required by the parser.

The second section contains grammar rules. Each rule defines how a non-terminal symbol can be expanded into other symbols. Actions written in C can also be attached to these rules.

The third section contains user-defined code such as the main() function and error-handling routines.

Consider the following grammar rule:

```
E : E '+' T   { printf("Addition\n"); }
  | T
  ;
```

This rule defines an expression E. The first production states that an expression can be formed by combining another expression with a term using the plus operator. The action prints "Addition" when this rule is applied.

The second production shows that an expression can also be a single term.

This example demonstrates how YACC combines grammar definitions with actions.

13.6 YACC Examples

```
A simple YACC example is shown below:

%%

E : E '+' T
  | T
  ;
```

```
T : 'a'

  ;

%%
```

This grammar recognizes simple addition expressions.

The rule for E allows recursive expressions such as a + a or a + a + a. The rule for T defines a term as the symbol a.

Now consider the input:

```
a + a
```

The parser first identifies the first a as a term (T), which is then reduced to an expression (E). It then processes the + symbol and the second a, which again becomes a term.

Finally, the entire expression matches the production:

```
E → E + T
```

Thus, the parser accepts the input as valid. In this parsing process tokens are combined according to grammar rules.

13.7 LEX & YACC Integration

In a compiler, LEX and YACC work together to perform lexical and syntax analysis.

LEX generates tokens from the input, and YACC receives those tokens and checks whether they form a valid structure.

This interaction is shown in **Figure 13.3**, where the output of LEX is used as input for YACC.

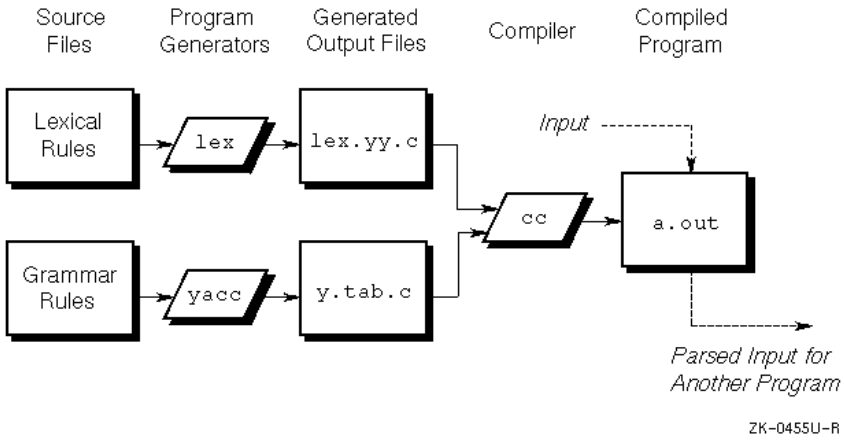


Figure 13.3: Integration of LEX and YACC

As shown in Figure 13.3, together they form the front-end of a compiler.

13.8 FLEX and BISON

FLEX and BISON are modern replacements for LEX and YACC. FLEX is used for lexical analysis, and BISON is used for syntax analysis.

They provide better performance, improved features, and enhanced compatibility with modern systems.

The relationship between these tools is illustrated in **Figure 13.4**, where FLEX and BISON perform the same roles as LEX and YACC.

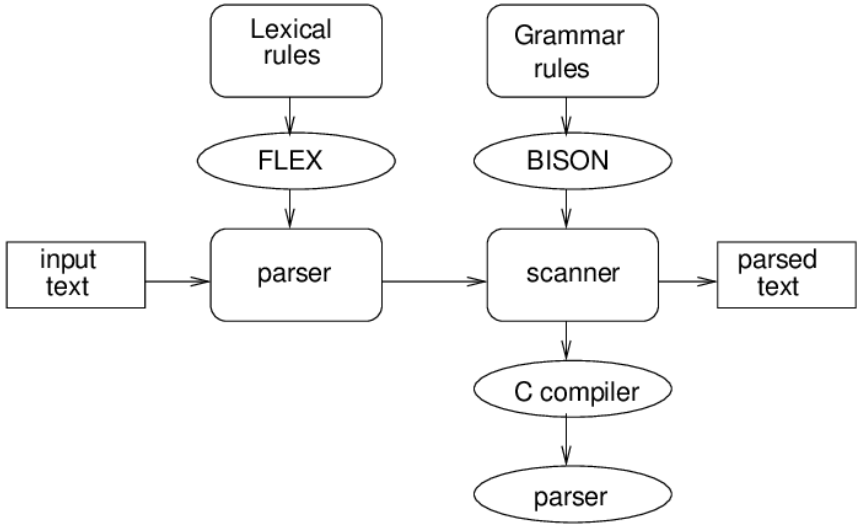


Figure 13.4: FLEX and BISON in Modern Compiler Design

EXERCISE

1. Write LEX rules for identifiers.
2. Write LEX rules for numbers.
3. Explain LEX structure with example.
4. Write YACC grammar for expressions.
5. Explain YACC parsing steps.
6. Integrate LEX and YACC.
7. Trace parsing using YACC.
8. Compare FLEX and LEX.
9. Explain BISON features.
10. Design parser using YACC.

CHAPTER 14

BUILDING A FULL COMPILER

14.1 Lexical Analyzer Implementation

The lexical analyzer is the first functional component of a compiler that processes the input program. Its task is to read the source code character by character and group them into meaningful tokens such as identifiers, numbers, and operators.

In a mini compiler, the lexical analyzer can be implemented using simple pattern checking. The input is scanned sequentially, and characters are classified into different token types.

Consider a simple example:

```
if (isalpha(ch))

    token = IDENTIFIER;

else if (isdigit(ch))

    token = NUMBER;
```

In this example, the function isalpha() checks whether the character is an alphabet, and isdigit() checks whether it is a digit. Based on the result, the token type is assigned.

The lexical analyzer reads characters and converts them into tokens for the next phase.

14.2 Parser Implementation

The parser takes tokens from the lexical analyzer and checks whether they follow the grammar rules of the language. In a mini

compiler, this can be implemented using a simple recursive descent approach.

Consider a basic grammar:

```
stmt → id = expr

expr → expr + term | term

term → id | num
```

A simplified parsing logic can be written using functions corresponding to each non-terminal.

```
void stmt() {

    match(ID);

    match('=');

    expr();

}
```

In this function, match() checks whether the current token matches the expected token. If it does not match, an error is reported.

The parser uses grammar rules to verify the structure of the input.

14.3 Integrating Lexical + Parser

In a full compiler, the lexical analyzer and parser must work together. The lexical analyzer produces tokens, and the parser consumes those tokens to check syntax.

This integration is achieved by connecting the output of the lexical analyzer directly to the input of the parser.

For example, consider the input:

```
a = b + 7;
```

The lexical analyzer produces tokens such as ID, ASSIGN, ID, PLUS, and NUM. These tokens are then passed to the parser, which verifies whether they match the grammar.

The interaction between these two components is illustrated in **Figure 14.1**, where tokens flow from the scanner to the parser.

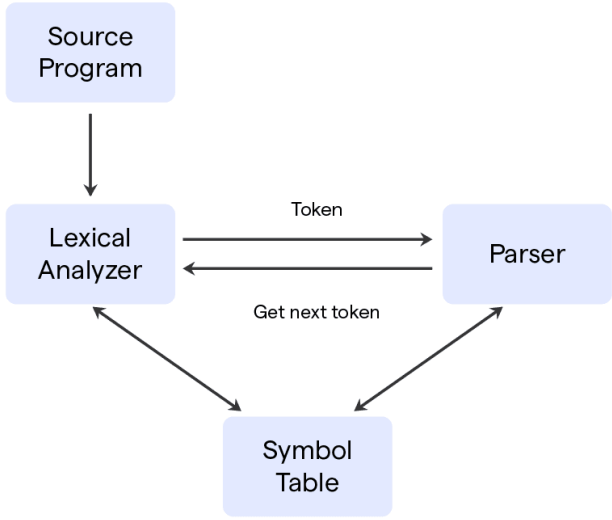


Figure 14.1: Integration of Lexical Analyzer and Parser

As shown in Figure 14.1, the parser depends on tokens generated by the lexical analyzer to perform syntax checking.

14.4 Handling Multiple Statements

A complete compiler must handle multiple statements rather than a single expression. This requires extending the grammar to support a sequence of statements.

Consider the following input:

```
x = y + 5;
```

```
z = x * 2;
```

To handle multiple statements, the grammar can be extended as:

```
program → stmt ; program | stmt
```

This grammar allows a sequence of statements separated by semicolons.

The parser processes each statement one by one. After completing one statement, it moves to the next statement until all input is processed.

14.5 Compiler Limitations and Improvements

A mini compiler is mainly developed to understand the fundamental concepts of compiler design. While it is very useful for learning purposes, it has several limitations when compared to a full-scale compiler used in real-world systems. These limitations arise because a mini compiler focuses on basic functionality and avoids handling complex programming features.

One major limitation is the **restricted support for programming constructs**. A mini compiler usually handles simple statements such as variable declarations and arithmetic expressions. However, it often fails to process more advanced constructs like loops, conditional statements, and functions. For example:

```
for(i = 0; i < 10; i++)
```

This type of statement requires more advanced grammar rules and proper semantic checking, which are often not included in a basic compiler.

Another limitation is related to **error handling**. Mini compilers typically provide only basic error detection and may stop execution

after encountering the first error. The error messages are often not descriptive, making it difficult for users to identify and correct mistakes efficiently.

In addition, mini compilers usually lack **optimization capabilities**. The generated code may contain unnecessary operations and may not be efficient in terms of execution time or memory usage. Similarly, the **code generation phase** is often simplified and does not consider machine-specific optimizations such as register allocation.

Mini compilers are also not suitable for handling large and complex programs, as they generally do not support modular programming, linking, or external libraries.

Summary of Limitations

- Limited support for complex constructs such as loops and functions
- Basic and sometimes unclear error handling
- Lack of optimization techniques
- Simplified code generation
- Not suitable for large-scale programs

To overcome these limitations, a compiler can be gradually improved by enhancing its features and capabilities. The development process usually follows a step-by-step approach, where new functionalities are added over time.

For instance, support for control structures such as **if-else statements and loops** can be introduced by extending the grammar rules. Error handling can be improved by generating detailed messages with proper descriptions and line numbers. Similarly, optimization techniques can be added to improve the efficiency of the generated code.

Possible Improvements

- Add support for control structures (if, for, while)
- Improve error reporting with clear messages
- Introduce optimization techniques
- Enhance code generation for better efficiency
- Extend support for functions and data structures

In conclusion, a mini compiler provides a strong foundation for understanding compiler design, but it has several limitations in terms of functionality and efficiency. By gradually adding advanced features such as improved error handling, optimization, and support for complex constructs, the compiler can evolve into a more complete and practical system capable of handling real-world programming requirements.

EXERCISE

1. Design lexical analyzer.
2. Implement parser.
3. Integrate lexical + parser.
4. Handle multiple statements.
5. Extend grammar for programs.
6. Trace full compilation.
7. Identify limitations.
8. Propose improvements.
9. Design full compiler architecture.
10. Explain complete workflow.

REFERENCES

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.
Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education.

[2] Srivastava, A., & Srivastava, M.
Compiler Design, Oxford University Press.

[3] Tremblay, J. P., & Sorenson, P. G.
The Theory and Practice of Compiler Writing, McGraw-Hill.

[4] Grune, D., Bal, H. E., Jacobs, C. J., & Langendoen, K.
Modern Compiler Design, Springer.

[5] Appel, A. W.
Modern Compiler Implementation in C, Cambridge University Press.

[6] Holub, A. I.
Compiler Design in C, Prentice Hall.

[7] Muchnick, S. S.
Advanced Compiler Design and Implementation, Morgan Kaufmann.

Online Resources

[8] GeeksforGeeks
https://www.geeksforgeeks.org

[9] TutorialsPoint
https://www.tutorialspoint.com

[10] GNU Flex & Bison Documentation
https://www.gnu.org/software

[11] Wikipedia – Compiler Design Topics
https://www.wikipedia.org

The concepts and examples presented in this book are based on standard compiler design literature and widely accepted academic resources.

About This Book

Compiler Design is a beginner-friendly book that explains the core concepts of compiler design in a clear and practical way. It covers topics from lexical and syntax analysis to code generation and optimization, using simple explanations and examples.

The book also includes practical implementations and a mini compiler project, helping readers understand how theory is applied in practice. Designed for students and beginners, this book provides a strong foundation for learning how programming languages are processed and executed.